PyTorch Introduction
A Stroll Through the Zoo of PyTorch

David Rotermund & Udo Ernst

PyTorch

PyTorch is a machine learning library based on the Torch library,[4][5][6] used for applications such as computer vision and natural language processing,[7] originally developed by Meta AI and now part of the Linux Foundation umbrella.[8][9][10][11] It is one of the most popular deep learning frameworks, alongside others such as TensorFlow,[12] offering free and open-source software released under the modified BSD license. Although the Python interface is more polished and the primary focus of development, PyTorch also has a C++ interface.[13]

A number of pieces of deep learning software are built on top of PyTorch, including Tesla Autopilot,[14] Uber's Pyro,[15] Hugging Face's Transformers,[16][17] and Catalyst.[18][19]

PyTorch provides two high-level features:[20]

- Tensor computing (like NumPy) with strong acceleration via graphics processing units (GPU)
- Deep neural networks built on a tape-based automatic differentiation system

Check here for an installation configurator.

Installing PyTorch

- **Windows**
  - *CPU only*

```
pip3 install torch torchvision torchaudio torchtext
```

  - *NVidia GPU*

```
pip3 install torch torchvision torchaudio torchtext  --index-url https://download.pytorch.org/whl/cu121
```

- **Linux**
  - *CPU only*

```
cd /home/[YOURUSERNAME]/P3.12/bin
./pip3 install torch torchvision torchaudio torchtext --index-url https://download.pytorch.org/whl/cpu
```

  - *NVidia GPU*

```
cd /home/[YOURUSERNAME]/P3.12/bin
./pip3 install torch torchvision torchaudio torchtext
```

PyTorch Packages

Package	
torch	PyTorch is a Python package that provides two high-level features: a. Tensor computation (like NumPy) with strong GPU acceleration b. Deep neural networks built on a tape-based autograd system
torchvision	The torchvision package consists of popular datasets, model architectures, and common image transformations for computer vision.
torchaudio	The aim of torchaudio is to apply PyTorch to the audio domain.
torchtext	Models, data loaders and abstractions for language processing, powered by PyTorch

Recommendation

Open Book: Dive into Deep Learning

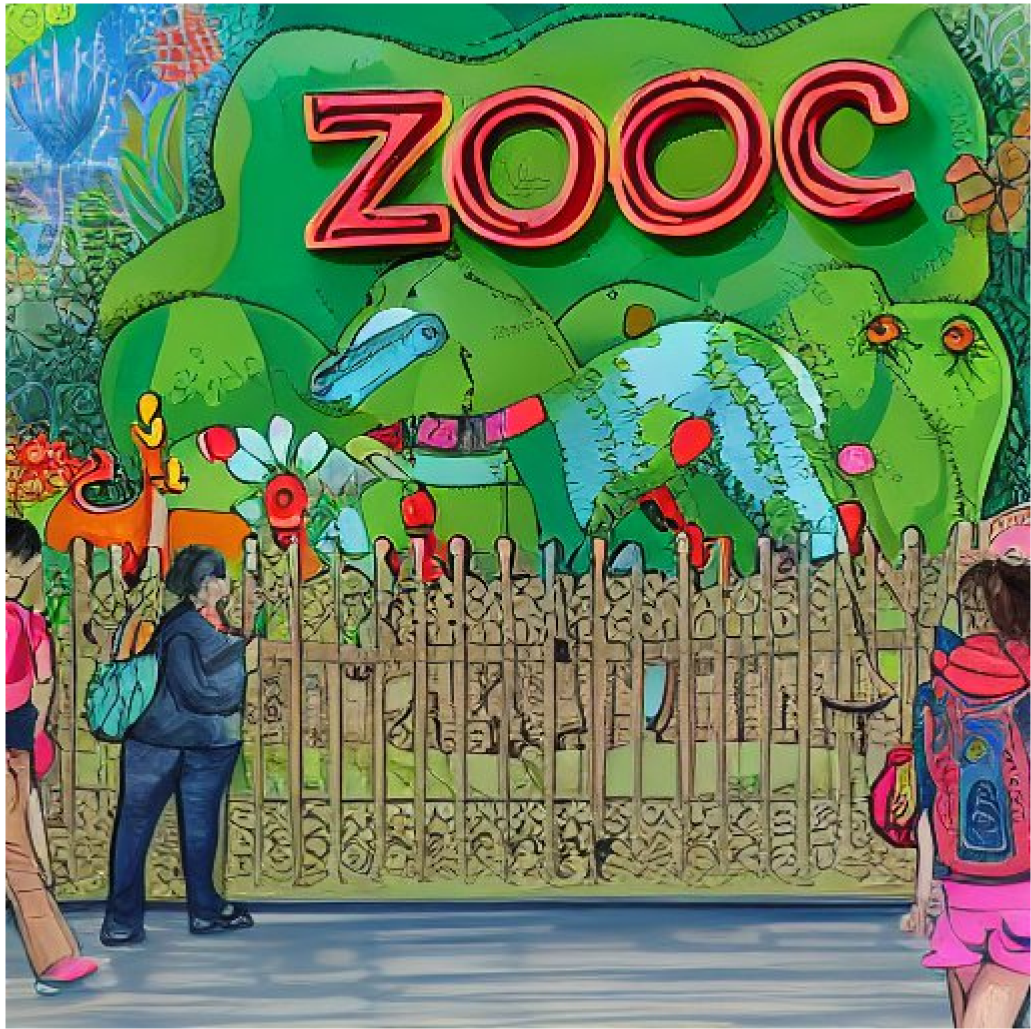
- ASTON ZHANG
- ZACHARY C. LIPTON
- MU LI
- ALEXANDER J. SMOLA

https://d2l.ai/d2l-en.pdf

Fig. 1.3.3 A donkey, a dog, a cat, and a rooster.

Welcome to the PyTorch Zoo!

I asked our local AI to give us a tour through the PyTorch Layer Zoo....



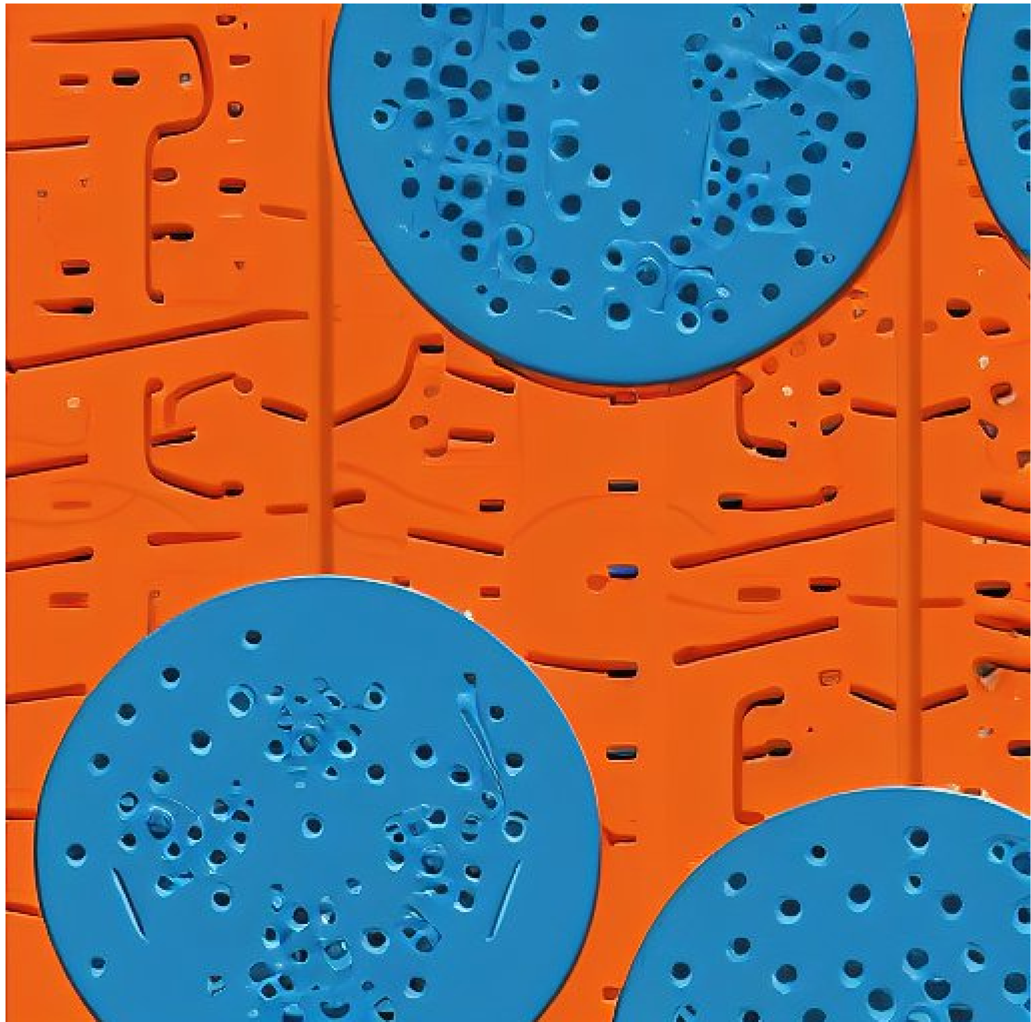
Layers are the basic building blocks in PyTorch

- A network is a combination of layers.
- When ever we can, we should use existing layers types.
- We can make our own layers.
- Layers are in **torch.nn.** and the name of layer **starts with an upper case letter**.

Let us look what is available and what is **relevant**.

The Foundation - Module

The journey begins at the foundation of PyTorch's neural network architecture: Module. This ancestral class is the root of all layers, providing the basic structure and functionality that enables the creation of complex neural networks. Just as a strong foundation is essential for building a sturdy structure, Module lays the groundwork for the various layers and modules that follow, including linear layers, convolutional layers, and more. Explore the fundamental components of Module, such as parameters, buffers, and hooks, to understand how they facilitate the construction of diverse neural network architectures.



Existing layers - Containers

● = relevant

●	Module	Base class for all neural network modules.

Sequential Containers - The Snake Pit

The next exhibit takes you to the Snake Pit, where a winding snake-like structure represents the concept of sequential containers in Python. Just as a snake grows and shrinks as it consumes its prey, a sequential container can dynamically add or remove elements, allowing for efficient storage and manipulation of data. Explore the different segments of the snake, each representing a single element in the container, and learn how indexing and slicing enable you to access and manipulate the data with ease.

Existing layers - Containers

● = relevant

●	Sequential	A sequential container.
	ModuleList	Holds submodules in a list.
	ModuleDict	Holds submodules in a dictionary.
	ParameterList	Holds parameters in a list.
	ParameterDict	Holds parameters in a dictionary.

Linear Layer - The Lion

The first exhibit features the Linear Layer, represented by a regal lion. Just as a lion's mane connects its head to its body, the Linear Layer connects every input to every output. The lion's strength and agility symbolize the layer's ability to tackle complex tasks.

From “Dive into Deep Learning”

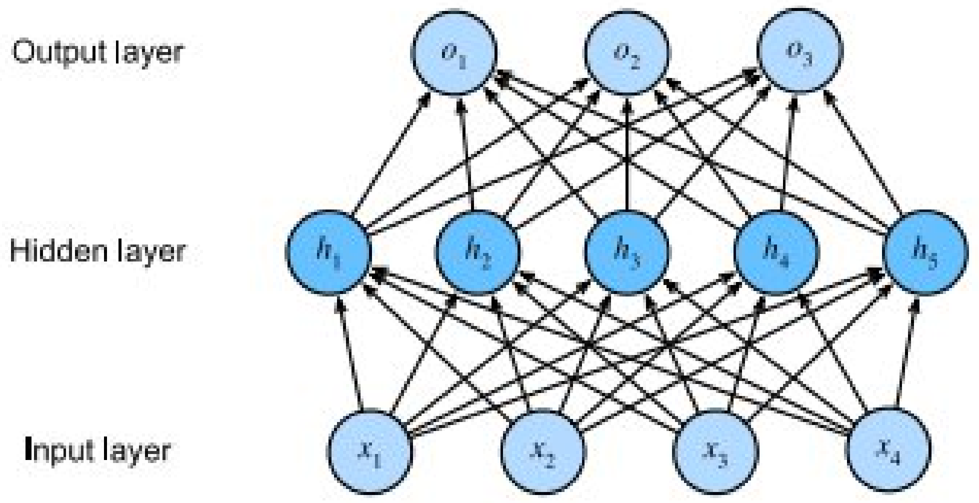


Fig. 5.1.1 An MLP with a hidden layer of five hidden units.

Existing layers - Linear Layers

● = relevant

	torch.nn.Identity	A placeholder identity operator that is argument-insensitive
●	torch.nn.Linear	Applies a linear transformation to the incoming data
	torch.nn.Bilinear	Applies a bilinear transformation to the incoming data
	torch.nn.LazyLinear	A torch.nn.Linear module where in_features is inferred.

Convolutional Layer - The Cheetah

Next, you'll visit the Convolutional Layer exhibit, featuring a swift cheetah. The cheetah's speed and agility represent the layer's ability to quickly scan and process data, using filters to extract features like a cheetah uses its speed to catch prey.

From “Dive into Deep Learning”

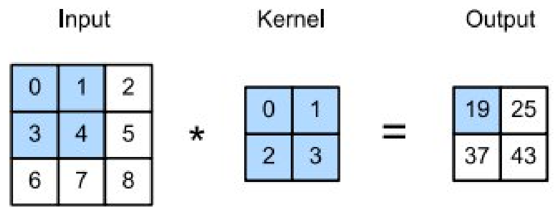


Fig. 7.2.1 Two-dimensional cross-correlation operation. The shaded portions are the first output element as well as the input and kernel tensor elements used for the output computation: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$.

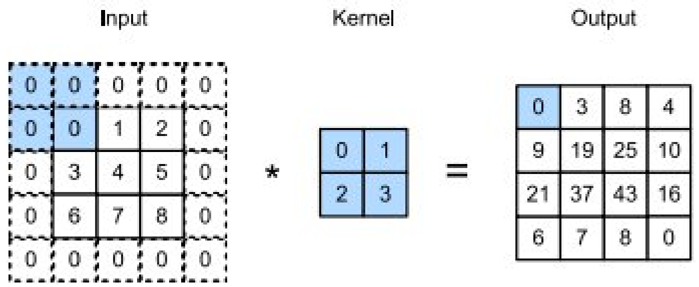


Fig. 7.3.2 Two-dimensional cross-correlation with padding.

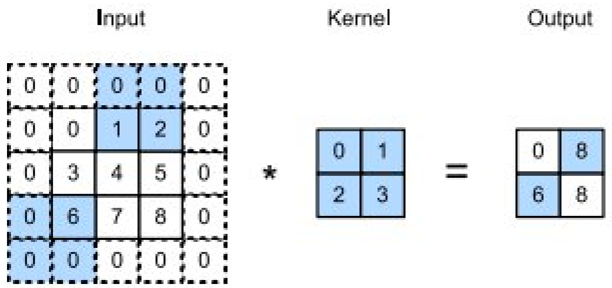


Fig. 7.3.3 Cross-correlation with strides of 3 and 2 for height and width, respectively.

Existing layers - Convolution Layers

● = relevant

●	torch.nn.Conv1d	Applies a 1D convolution over an input signal composed of several input planes.
●	torch.nn.Conv2d	Applies a 2D convolution over an input signal composed of several input planes.
●	torch.nn.Conv3d	Applies a 3D convolution over an input signal composed of several input planes.
	torch.nn.ConvTranspose1d	Applies a 1D transposed convolution operator over an input image composed of several input planes.
	torch.nn.ConvTranspose2d	Applies a 2D transposed convolution operator over an input image composed of several input planes.
	torch.nn.ConvTranspose3d	Applies a 3D transposed convolution operator over an input image composed of several input planes.
	torch.nn.LazyConv1d	A torch.nn.Conv1d module with lazy initialization of the in_channels argument of the Conv1d that is inferred from the input.size(1).
	torch.nn.LazyConv2d	A torch.nn.Conv2d module with lazy initialization of the in_channels argument of the Conv2d that is inferred from the input.size(1).
	torch.nn.LazyConv3d	A torch.nn.Conv3d module with lazy initialization of the in_channels argument of the Conv3d that is inferred from the input.size(1).
	torch.nn.LazyConvTranspose1d	A torch.nn.ConvTranspose1d module with lazy initialization of the in_channels argument of the ConvTranspose1d that is inferred from the input.size(1).
	torch.nn.LazyConvTranspose2d	A torch.nn.ConvTranspose2d module with lazy initialization of the in_channels argument of the ConvTranspose2d that is inferred from the input.size(1).
	torch.nn.LazyConvTranspose3d	A torch.nn.ConvTranspose3d module with lazy initialization of the in_channels argument of the ConvTranspose3d that is inferred from the input.size(1).
	torch.nn.Unfold	Extracts sliding local blocks from a batched input tensor.
	torch.nn.Fold	Combines an array of sliding local blocks into a large containing tensor.

I use those for my spiking convolution networks

Pooling Layers - The Koi Pond

The Pooling Layer is represented by a serene koi pond, where fish swim together in harmony. Just as the fish reduce the complexity of the water's surface by swimming in schools, the Pooling Layer reduces the spatial dimensions of the input data, helping the model to focus on the most important features.

From “Dive into Deep Learning”

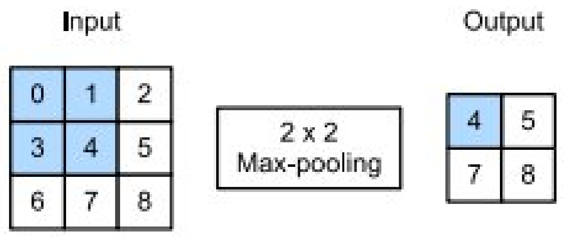


Fig. 7.5.1 Max-pooling with a pooling window shape of 2×2. The shaded portions are the first output element as well as the input tensor elements used for the output computation: $\max(0, 1, 3, 4) = 4$.

Existing layers - Pooling layers

● = relevant

●	torch.nn.MaxPool1d	Applies a 1D max pooling over an input signal composed of several input planes.
●	torch.nn.MaxPool2d	Applies a 2D max pooling over an input signal composed of several input planes.
●	torch.nn.MaxPool3d	Applies a 3D max pooling over an input signal composed of several input planes.
	torch.nn.MaxUnpool1d	Computes a partial inverse of MaxPool1d.
	torch.nn.MaxUnpool2d	Computes a partial inverse of MaxPool2d.
	torch.nn.MaxUnpool3d	Computes a partial inverse of MaxPool3d.
●	torch.nn.AvgPool1d	Applies a 1D average pooling over an input signal composed of several input planes.
●	torch.nn.AvgPool2d	Applies a 2D average pooling over an input signal composed of several input planes.
●	torch.nn.AvgPool3d	Applies a 3D average pooling over an input signal composed of several input planes.
	torch.nn.FractionalMaxPool2d	Applies a 2D fractional max pooling over an input signal composed of several input planes.
	torch.nn.FractionalMaxPool3d	Applies a 3D fractional max pooling over an input signal composed of several input planes.
	torch.nn.LPPool1d	Applies a 1D power-average pooling over an input signal composed of several input planes.
	torch.nn.LPPool2d	Applies a 2D power-average pooling over an input signal composed of several input planes.
	torch.nn.AdaptiveMaxPool1d	Applies a 1D adaptive max pooling over an input signal composed of several input planes.
	torch.nn.AdaptiveMaxPool2d	Applies a 2D adaptive max pooling over an input signal composed of several input planes.
	torch.nn.AdaptiveMaxPool3d	Applies a 3D adaptive max pooling over an input signal composed of several input planes.
	torch.nn.AdaptiveAvgPool1d	Applies a 1D adaptive average pooling over an input signal composed of several input planes.
	torch.nn.AdaptiveAvgPool2d	Applies a 2D adaptive average pooling over an input signal composed of several input planes.
	torch.nn.AdaptiveAvgPool3d	Applies a 3D adaptive average pooling over an input signal composed of several input planes.

Padding Layers - The Framed Artwork

The Padding Layer is represented by a beautifully framed artwork, where a delicate border surrounds a stunning painting. Just as the frame enhances and protects the artwork, the Padding Layer adds zeros to the input data, helping to preserve important information at the borders and improving model performance.

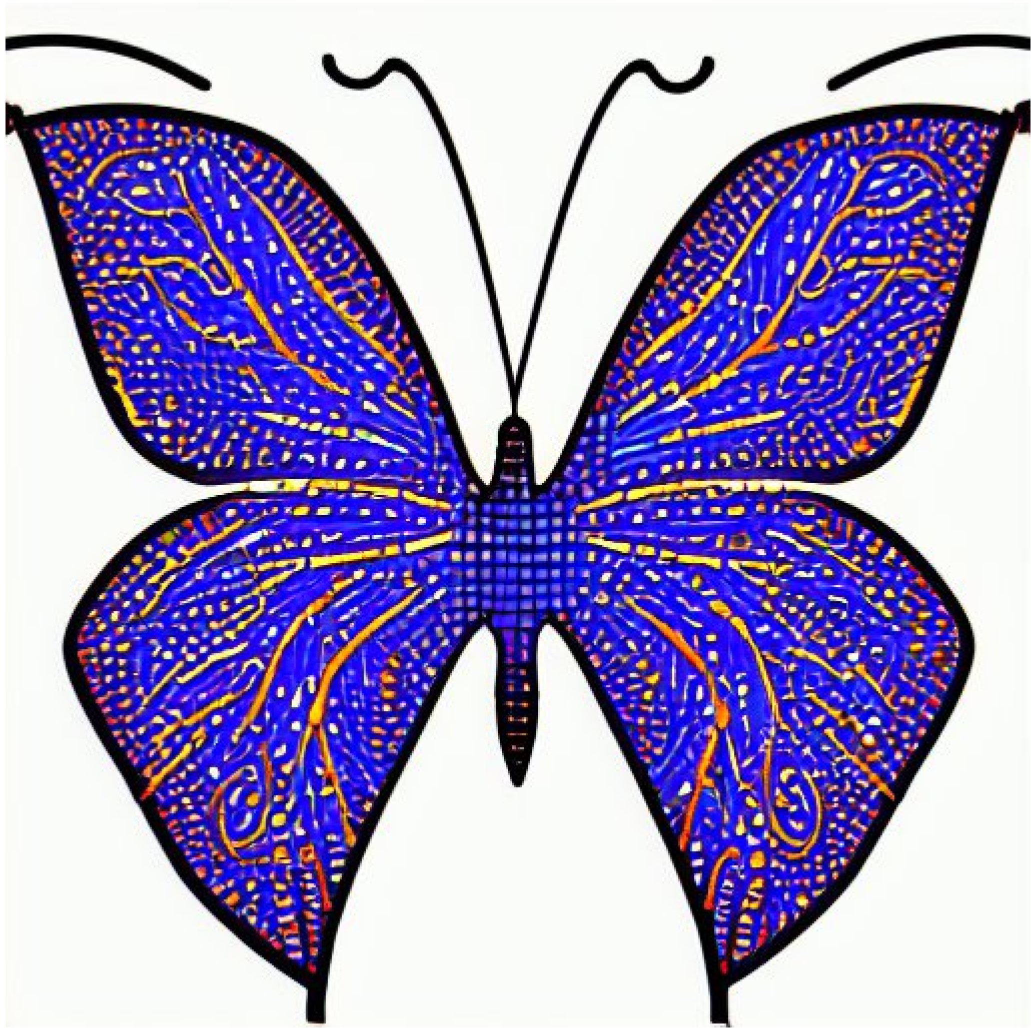
Existing layers - Padding Layers

● = relevant

torch.nn.ReflectionPad1d	Pads the input tensor using the reflection of the input boundary.
torch.nn.ReflectionPad2d	Pads the input tensor using the reflection of the input boundary.
torch.nn.ReflectionPad3d	Pads the input tensor using the reflection of the input boundary.
torch.nn.ReplicationPad1d	Pads the input tensor using replication of the input boundary.
torch.nn.ReplicationPad2d	Pads the input tensor using replication of the input boundary.
torch.nn.ReplicationPad3d	Pads the input tensor using replication of the input boundary.
torch.nn.ZeroPad1d	Pads the input tensor boundaries with zero.
torch.nn.ZeroPad2d	Pads the input tensor boundaries with zero.
torch.nn.ZeroPad3d	Pads the input tensor boundaries with zero.
torch.nn.ConstantPad1d	Pads the input tensor boundaries with a constant value.
torch.nn.ConstantPad2d	Pads the input tensor boundaries with a constant value.
torch.nn.ConstantPad3d	Pads the input tensor boundaries with a constant value.

Activation Function Layers - The Butterfly Garden

The final exhibit features a beautiful butterfly garden, representing the various activation functions like ReLU, Sigmoid, and Tanh. Just as different butterflies have unique characteristics and behaviors, each activation function introduces non-linearity into the model in its own way, allowing it to learn complex patterns.



From “Dive into Deep Learning”

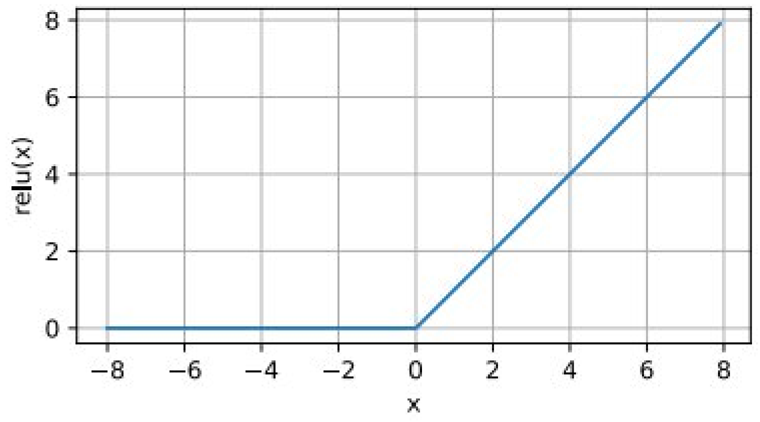
ReLU Function

The most popular choice, due to both simplicity of implementation and its good performance on a variety of predictive tasks, is the *rectified linear unit* (*ReLU*) (Nair and Hinton, 2010). ReLU provides a very simple nonlinear transformation. Given an element x, the function is defined as the maximum of that element and 0:

$$\mathrm{ReLU}(x) = \max(x, 0). \tag{5.1.4}$$

Informally, the ReLU function retains only positive elements and discards all negative elements by setting the corresponding activations to 0. To gain some intuition, we can plot the function. As you can see, the activation function is piecewise linear.

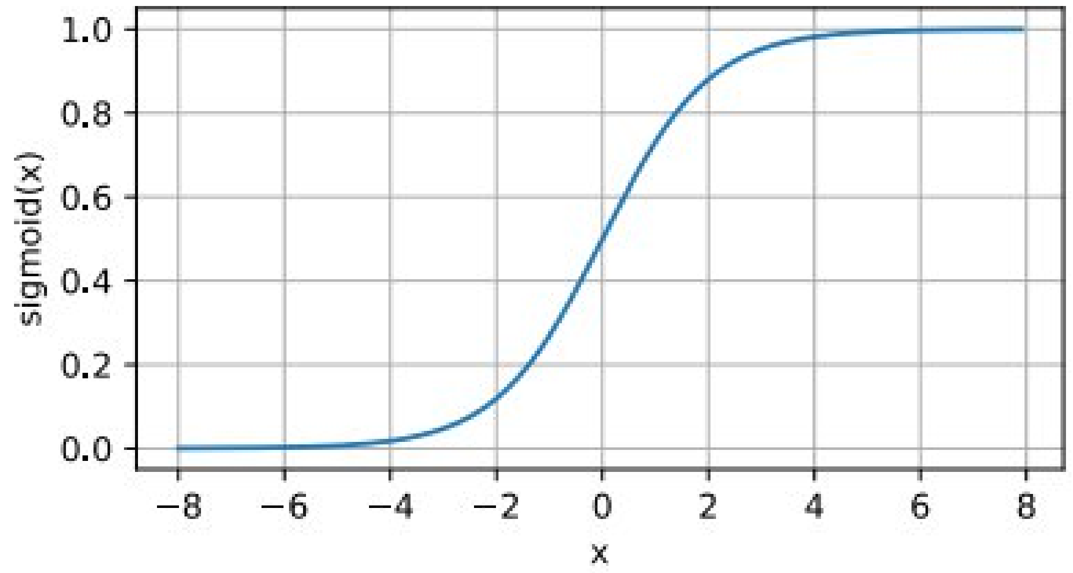
```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



From “Dive into Deep Learning”

Below, we plot the sigmoid function. Note that when the input is close to 0, the sigmoid function approaches a linear transformation.

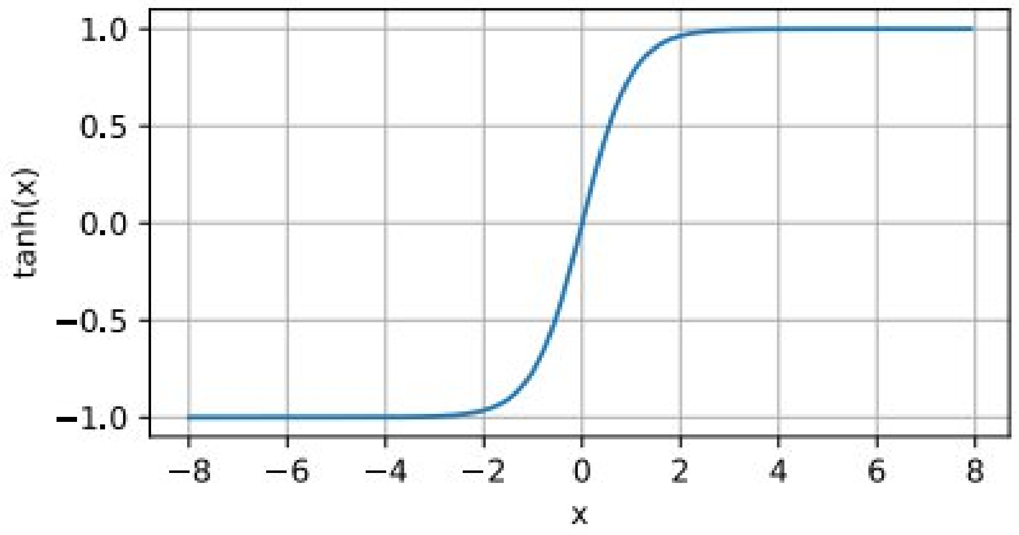
```
y = torch.sigmoid(x)
d2l.plot(x.detach(), y.detach(), 'x', 'sigmoid(x)', figsize=(5, 2.5))
```



From “Dive into Deep Learning”

We plot the tanh function below. Note that as input nears 0, the tanh function approaches a linear transformation. Although the shape of the function is similar to that of the sigmoid function, the tanh function exhibits point symmetry about the origin of the coordinate system (Kalman and Kwasny, 1992).

```
y = torch.tanh(x)
d2l.plot(x.detach(), y.detach(), 'x', 'tanh(x)', figsize=(5, 2.5))
```



Existing layers - Non-linear Activations

(weighted sum, nonlinearity)

● = relevant

	torch.nn.ELU	Applies the Exponential Linear Unit (ELU) function, element-wise, as described in the paper: Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs).
	torch.nn.Hardshrink	Applies the Hard Shrinkage (Hardshrink) function element-wise.
	torch.nn.Hardsigmoid	Applies the Hardsigmoid function element-wise.
	torch.nn.Hardtanh	Applies the HardTanh function element-wise.
	torch.nn.Hardswish	Applies the Hardswish function, element-wise, as described in the paper: Searching for MobileNetV3.
●	torch.nn.LeakyReLU	Applies the element-wise function:
	torch.nn.LogSigmoid	Applies the element-wise function:
	torch.nn.MultiheadAttention	Allows the model to jointly attend to information from different representation subspaces as described in the paper: Attention Is All You Need.
	torch.nn.PReLU	Applies the element-wise function:
●	torch.nn.ReLU	Applies the rectified linear unit function element-wise:
	torch.nn.ReLU6	Applies the element-wise function:
	torch.nn.RReLU	Applies the randomized leaky rectified liner unit function, element-wise, as described in the paper:
	torch.nn.SELU	Applied element-wise…
	torch.nn.CELU	Applies the element-wise function…
	torch.nn.GELU	Applies the Gaussian Error Linear Units function:

Existing layers - Non-linear Activations

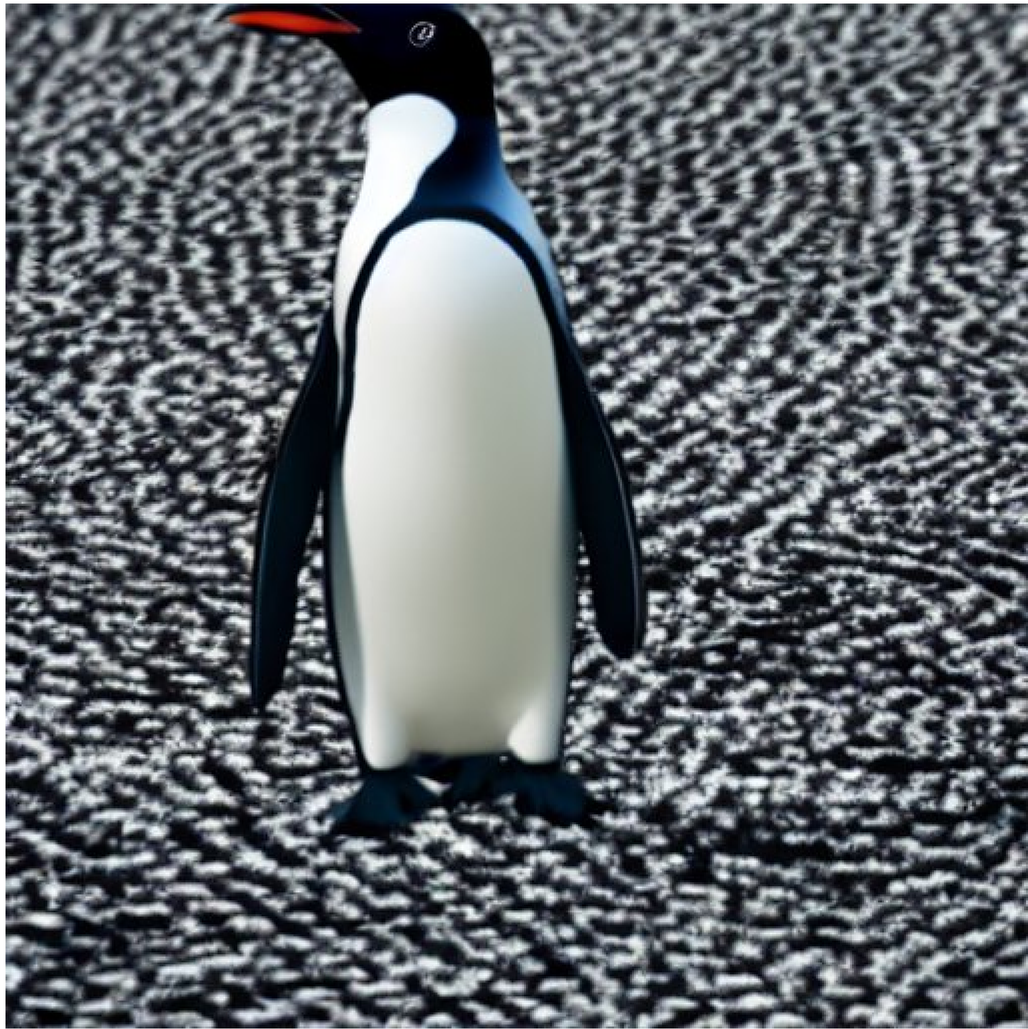
(weighted sum, nonlinearity)

● = relevant

●	torch.nn.Sigmoid	Applies the element-wise function:…
	torch.nn.SiLU	Applies the Sigmoid Linear Unit (SiLU) function, element-wise.
	torch.nn.Mish	Applies the Mish function, element-wise.
	torch.nn.Softplus	Applies the Softplus function
	torch.nn.Softshrink	Applies the soft shrinkage function elementwise:
	torch.nn.Softsign	Applies the element-wise function:…
●	torch.nn.Tanh	Applies the Hyperbolic Tangent (Tanh) function element-wise.
	torch.nn.Tanhshrink	Applies the element-wise function:
	torch.nn.Threshold	Thresholds each element of the input Tensor.
	torch.nn.GLU	Applies the gated linear unit function

Batch Normalization Layer - The Penguin

The Batch Normalization Layer is depicted as a penguin, which thrives in groups and follows norms. The layer normalizes the input data for each mini-batch, ensuring that all the inputs follow the same distribution and improving model stability.



Existing layers - Normalization Layers

● = relevant

●	torch.nn.BatchNorm1d	Applies Batch Normalization over a 2D or 3D input as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .
●	torch.nn.BatchNorm2d	Applies Batch Normalization over a 4D input (a mini-batch of 2D inputs with additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .
●	torch.nn.BatchNorm3d	Applies Batch Normalization over a 5D input (a mini-batch of 3D inputs with additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .
	torch.nn.LazyBatchNorm1d	A torch.nn.BatchNorm1d module with lazy initialization of the num_features argument of the BatchNorm1d that is inferred from the input.size(1).
	torch.nn.LazyBatchNorm2d	A torch.nn.BatchNorm2d module with lazy initialization of the num_features argument of the BatchNorm2d that is inferred from the input.size(1).
	torch.nn.LazyBatchNorm3d	A torch.nn.BatchNorm3d module with lazy initialization of the num_features argument of the BatchNorm3d that is inferred from the input.size(1).
	torch.nn.GroupNorm	Applies Group Normalization over a mini-batch of inputs as described in the paper Group Normalization
	torch.nn.SyncBatchNorm	Applies Batch Normalization over a N-Dimensional input (a mini-batch of [N-2]D inputs with additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .
	torch.nn.InstanceNorm1d	Applies Instance Normalization over a 2D (unbatched) or 3D (batched) input as described in the paper Instance Normalization: The Missing Ingredient for Fast Stylization.
	torch.nn.InstanceNorm2d	Applies Instance Normalization over a 4D input (a mini-batch of 2D inputs with additional channel dimension) as described in the paper Instance Normalization: The Missing Ingredient for Fast Stylization.
	torch.nn.InstanceNorm3d	Applies Instance Normalization over a 5D input (a mini-batch of 3D inputs with additional channel dimension) as described in the paper Instance Normalization: The Missing Ingredient for Fast Stylization.
	torch.nn.LazyInstanceNorm1d	A torch.nn.InstanceNorm1d module with lazy initialization of the num_features argument of the InstanceNorm1d that is inferred from the input.size(1).
	torch.nn.LazyInstanceNorm2d	A torch.nn.InstanceNorm2d module with lazy initialization of the num_features argument of the InstanceNorm2d that is inferred from the input.size(1).
	torch.nn.LazyInstanceNorm3d	A torch.nn.InstanceNorm3d module with lazy initialization of the num_features argument of the InstanceNorm3d that is inferred from the input.size(1).
●	torch.nn.LayerNorm	Applies Layer Normalization over a mini-batch of inputs as described in the paper Layer Normalization
	torch.nn.LocalResponseNorm	Applies local response normalization over an input signal composed of several input planes, where channels occupy the second dimension.

Recurrent Neural Network (RNN) Layer - The Elephant

The RNN Layer is depicted as an elephant, with its excellent memory and ability to recall past experiences. Just as an elephant never forgets, the RNN Layer uses feedback connections to keep track of information over time, making it perfect for sequential data.

From “Dive into Deep Learning”

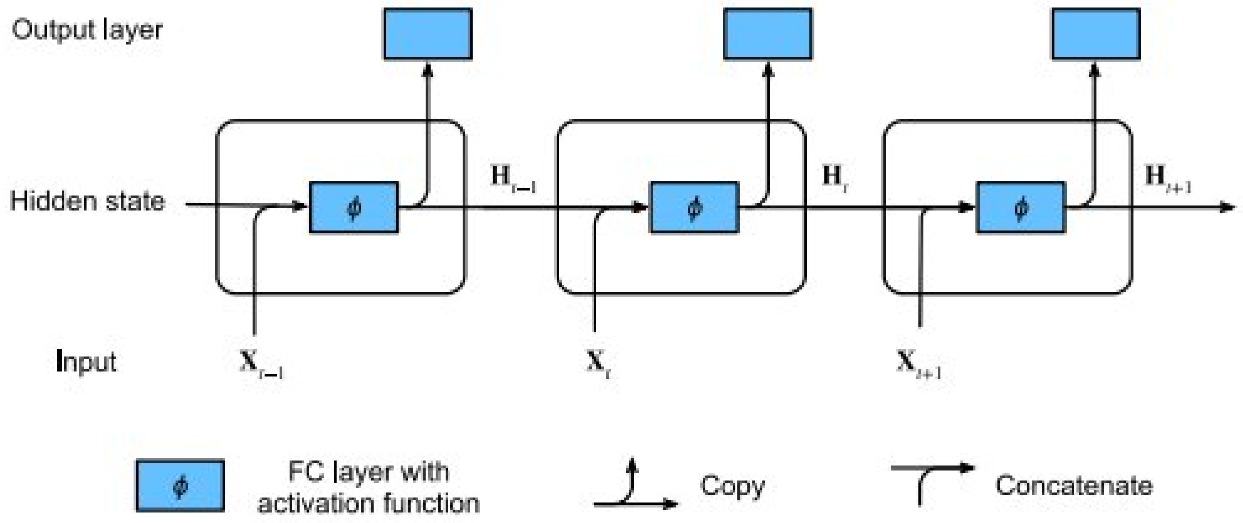
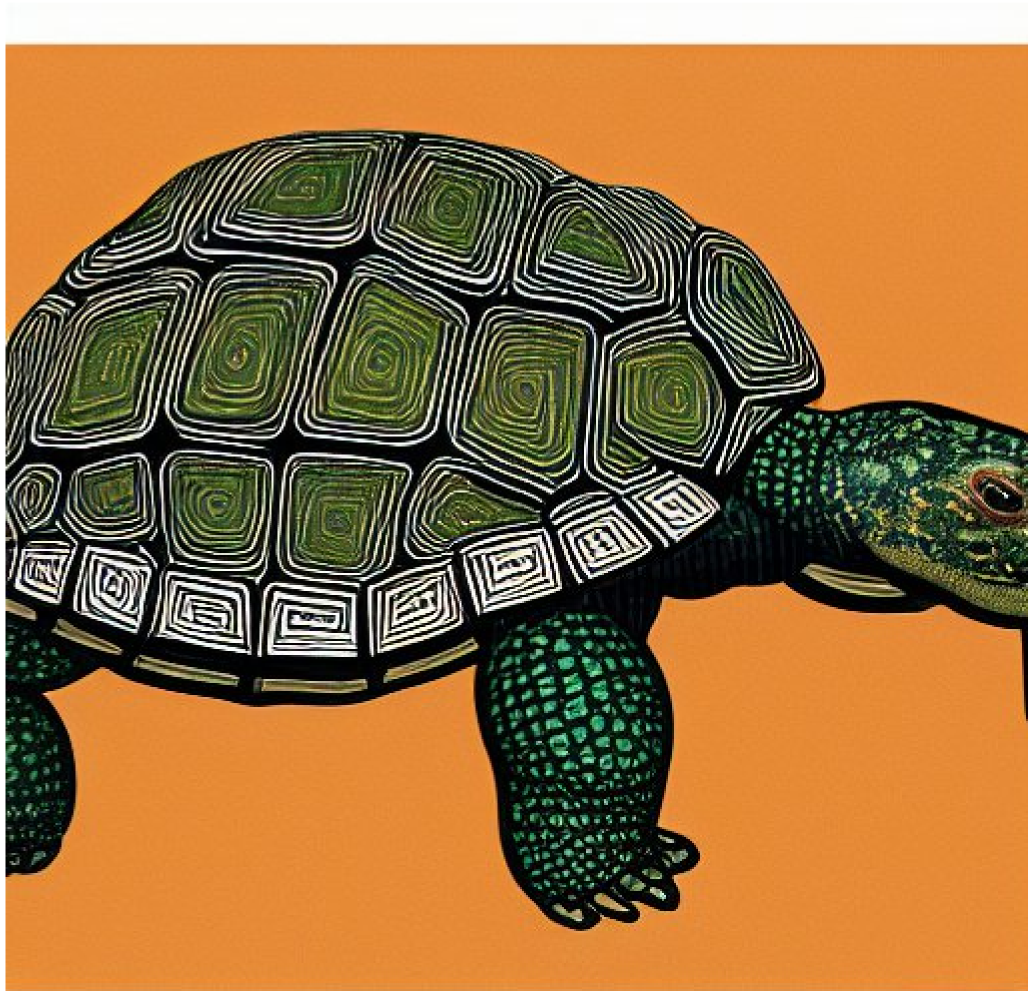


Fig. 9.4.1 An RNN with a hidden state.

Long Short-Term Memory (LSTM) Layer - The Tortoise

The LSTM Layer is represented by a tortoise, symbolizing its ability to carry information over long distances and remember it for extended periods. The tortoise's shell protects it from the outside world, just as the LSTM Layer's memory cells and gates protect the information from vanishing.



From “Dive into Deep Learning”

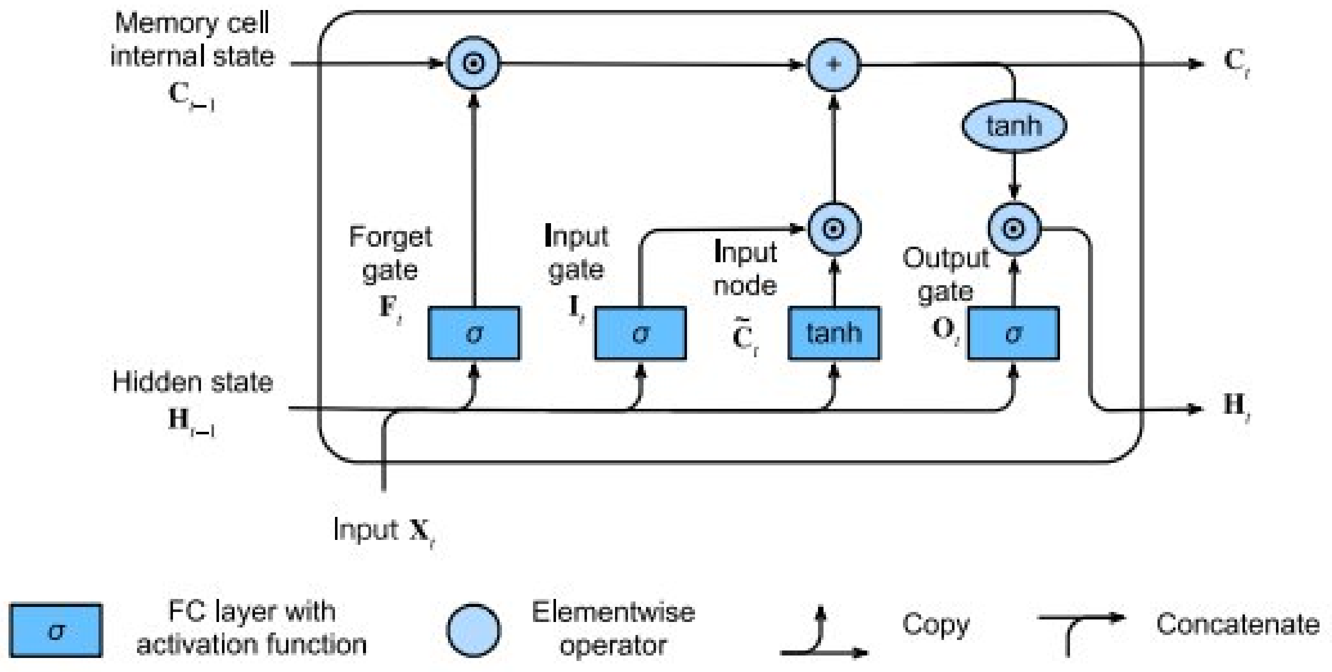

Memory cell internal state $\mathbf{C}_{t-1}$
$\mathbf{C}_t$
tanh
Forget gate $\mathbf{F}_t$
Input gate $\mathbf{I}_t$
Input node $\tilde{\mathbf{C}}_t$
Output gate $\mathbf{O}_t$
σ
σ
tanh
σ
Hidden state $\mathbf{H}_{t-1}$
$\mathbf{H}_t$
Input $\mathbf{X}_t$
FC layer with activation function
Elementwise operator
Copy
Concatenate

Fig. 10.1.4 Computing the hidden state in an LSTM model.

Existing layers - Recurrent Layers

● = relevant

RNN, GRU, LSTM and such lives here.
If you don’t know what this means then you don’t need them… It is for modelling time series.

	torch.nn.RNNBase	Base class for RNN modules (RNN, LSTM, GRU).
●	torch.nn.RNN	Applies a multi-layer Elman
●	torch.nn.LSTM	Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.
●	torch.nn.GRU	Applies a multi-layer gated recurrent unit (GRU) RNN to an input sequence.
	torch.nn.RNNCell	An Elman RNN cell with tanh or ReLU non-linearity.
	torch.nn.LSTMCell	A long short-term memory (LSTM) cell.
	torch.nn.GRUCell	A gated recurrent unit (GRU) cell

Transformer Layer - The Parrot

The Transformer Layer is depicted as a colorful parrot, known for its ability to mimic and understand context. The parrot's self-attention mechanisms allow it to weigh the importance of different input elements, just like a parrot weighs the importance of different sounds and words.

From “Dive into Deep Learning”

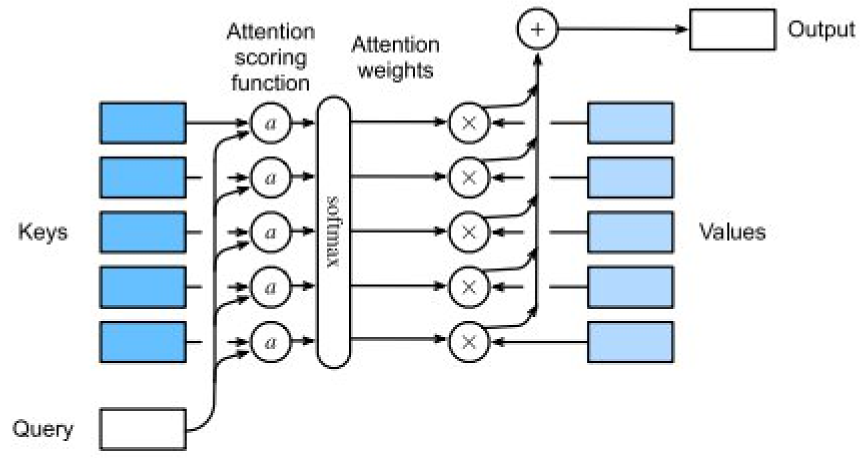


Fig. 11.3.1 Computing the output of attention pooling as a weighted average of values, where weights are computed with the attention scoring function a and the softmax operation.

$$\mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V} \in \mathbb{R}^{n\times v}. \tag{11.3.6}$$

Existing layers - Transformer Layers

If you do research on transformers or want to use them then you will write them yourself.

torch.nn.Transformer	A transformer model.
torch.nn.TransformerEncoder	TransformerEncoder is a stack of N encoder layers.
torch.nn.TransformerDecoder	TransformerDecoder is a stack of N decoder layers
torch.nn.TransformerEncoderLayer	TransformerEncoderLayer is made up of self-attn and feedforward network.
torch.nn.TransformerDecoderLayer	TransformerDecoderLayer is made up of self-attn, multi-head-attn and feedforward network.

Dropout Layer - The Chameleon

The Dropout Layer is represented by a chameleon, which can change its appearance to adapt to its surroundings. The Dropout Layer randomly sets a fraction of the output elements to zero during training, allowing the model to adapt and prevent overfitting.

From “Dive into Deep Learning”

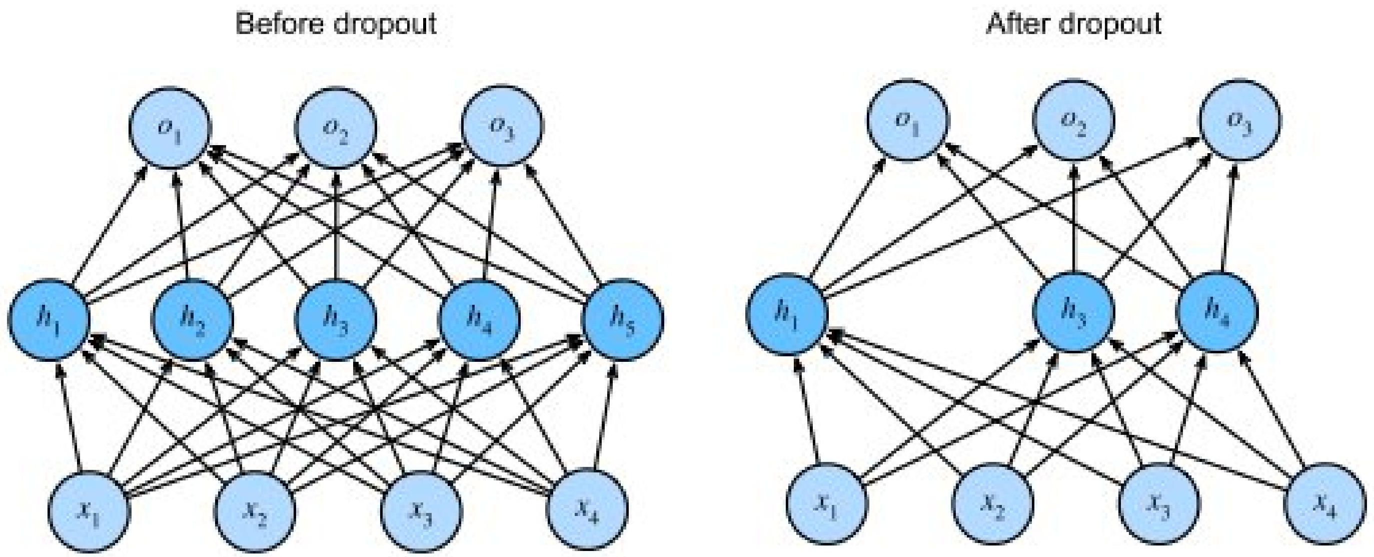


Fig. 5.6.1 MLP before and after dropout.

Existing layers - Dropout Layers

● = relevant

●	torch.nn.Dropout	During training, randomly zeroes some of the elements of the input tensor with probability p using samples from a Bernoulli distribution.
●	torch.nn.Dropout1d	Randomly zero out entire channels (a channel is a 1D feature map).
●	torch.nn.Dropout2d	Randomly zero out entire channels (a channel is a 2D feature map).
●	torch.nn.Dropout3d	Randomly zero out entire channels (a channel is a 3D feature map)
	torch.nn.AlphaDropout	Applies Alpha Dropout over the input.
	torch.nn.FeatureAlphaDropout	Randomly masks out entire channels (a channel is a feature map)

Sparse Layers - The Desert Landscape

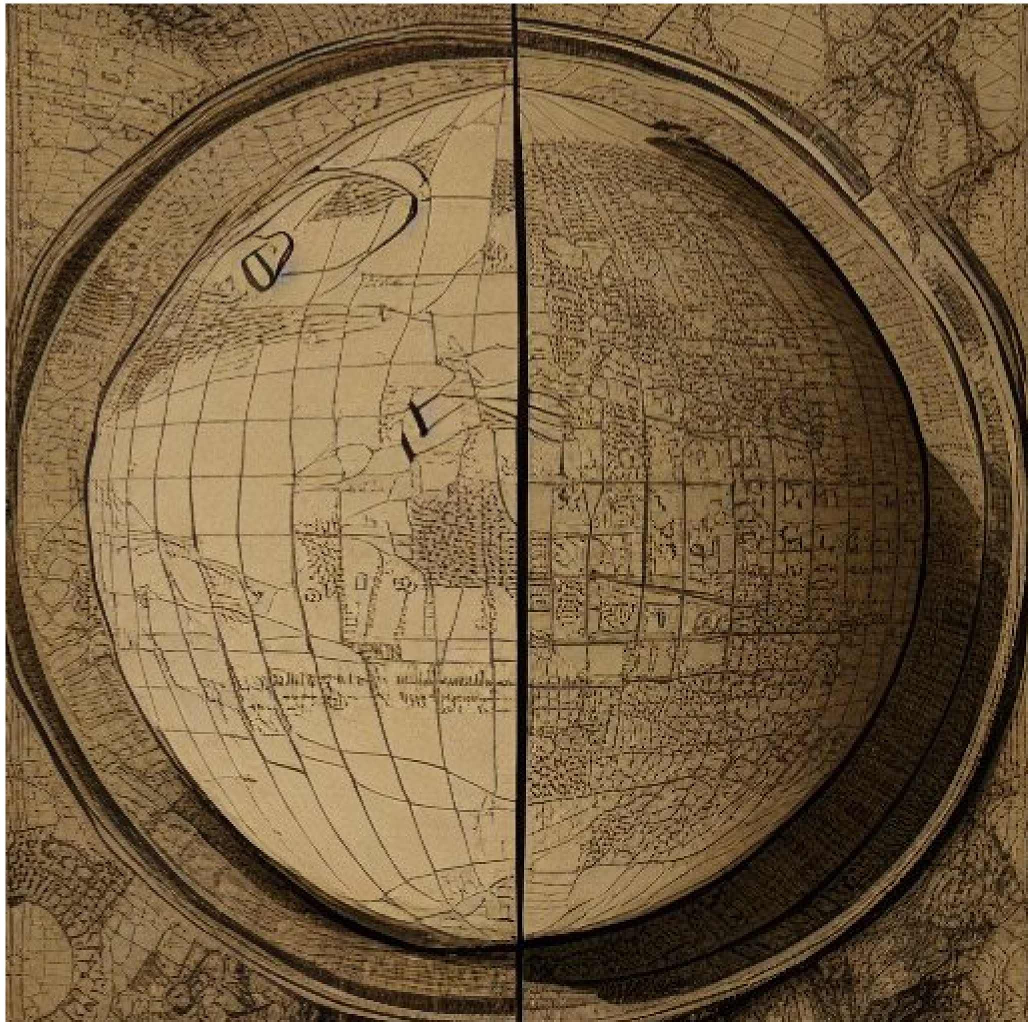
The Sparse Layer is represented by a vast and arid desert landscape, where cacti and succulents thrive in the harsh environment. Just as these plants have adapted to survive with limited resources, the Sparse Layer helps the model to focus on the most important features of the input data, eliminating unnecessary connections and improving efficiency.

Existing layers - Sparse Layers

torch.nn.Embedding	A simple lookup table that stores embeddings of a fixed dictionary and size.
torch.nn.EmbeddingBag	Computes sums or means of ‘bags’ of embeddings, without instantiating the intermediate embeddings.

Distance Functions - The Map Room

The Distance Function is represented by a map room, where cartographers measure distances between different locations. Just as the cartographers use various distance metrics to calculate the shortest path, the Distance Function measures the similarity or dissimilarity between input data points, helping the model to make predictions and classify examples.

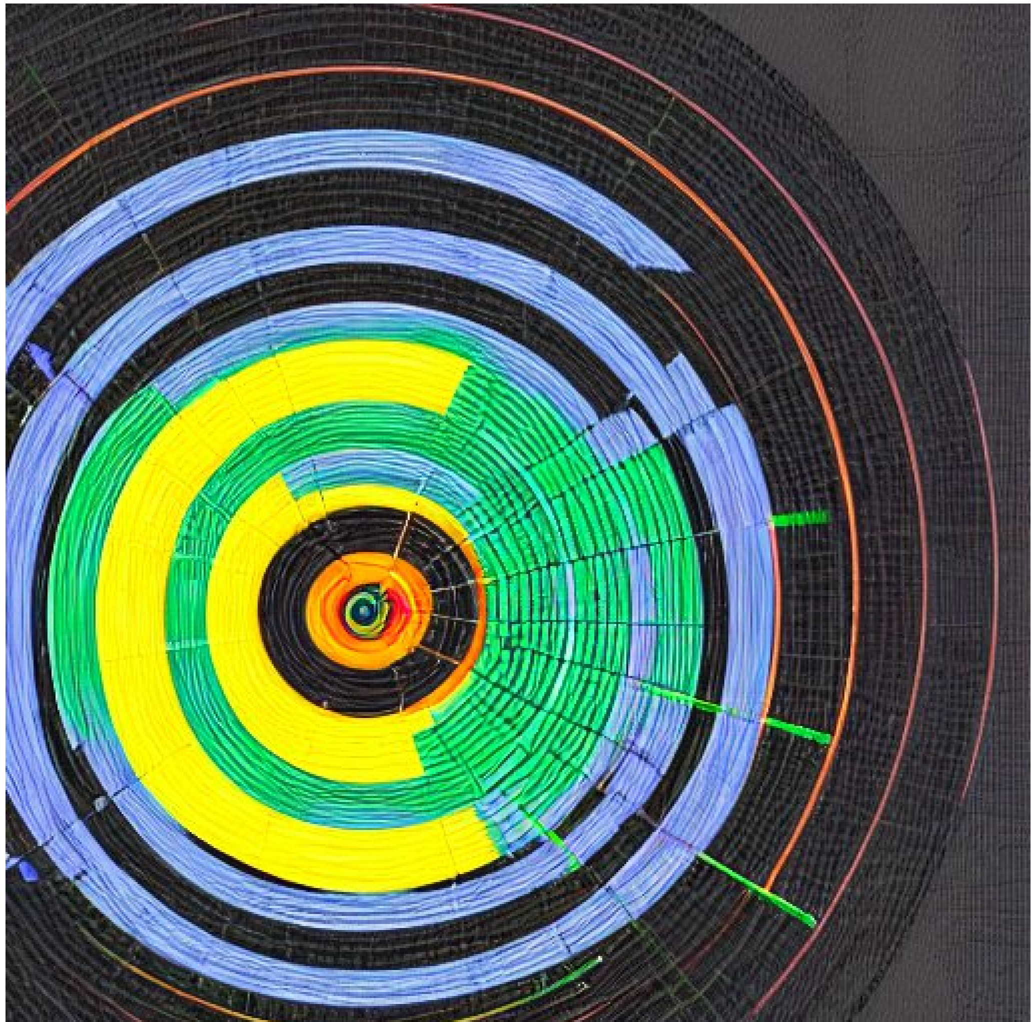


Existing layers - Distance Functions

torch.nn.CosineSimilarity	Returns cosine similarity
torch.nn.PairwiseDistance	Computes the pairwise distance between input vectors, or between columns of input matrices.

Loss Functions - The Target Practice Range

The Loss Function is represented by a target practice range, where archers aim to hit the bullseye. Just as the archers strive to minimize their distance from the target, the Loss Function measures the difference between the model's predictions and the actual labels, helping the model to learn and improve.



Existing layers - Loss Functions

● = relevant

	torch.nn.L1Loss	Creates a criterion that measures the mean absolute error (MAE) between each element in the input
●	torch.nn.MSELoss	Creates a criterion that measures the mean squared error (squared L2 norm) between each element in the input
●	torch.nn.CrossEntropyLoss	This criterion computes the cross entropy loss between input logits and target.
	torch.nn.CTCLoss	The Connectionist Temporal Classification loss.
	torch.nn.NLLLoss	The negative log likelihood loss.
	torch.nn.PoissonNLLLoss	Negative log likelihood loss with Poisson distribution of target.
	torch.nn.GaussianNLLLoss	Gaussian negative log likelihood loss.
	torch.nn.KLDivLoss	The Kullback-Leibler divergence loss.
	torch.nn.BCELoss	Creates a criterion that measures the Binary Cross Entropy between the target and the input probabilities:
	torch.nn.BCEWithLogitsLoss	This loss combines a Sigmoid layer and the BCELoss in one single class.
	torch.nn.MarginRankingLoss	Creates a criterion that measures the loss
	torch.nn.HingeEmbeddingLoss	Measures the loss given an input tensor
	torch.nn.MultiLabelMarginLoss	Creates a criterion that optimizes a multi-class multi-classification hinge loss (margin-based loss)
	torch.nn.HuberLoss	Creates a criterion that uses a squared term if the absolute element-wise error falls below delta and a delta-scaled L1 term otherwise.
	torch.nn.SmoothL1Loss	Creates a criterion that uses a squared term if the absolute element-wise error falls below beta and an L1 term otherwise.
	torch.nn.SoftMarginLoss	Creates a criterion that optimizes a two-class classification logistic loss
	torch.nn.MultiLabelSoftMarginLoss	Creates a criterion that optimizes a multi-label one-versus-all loss based on max-entropy
	torch.nn.CosineEmbeddingLoss	Creates a criterion that measures the loss given input tensors and a Tensor label
	torch.nn.MultiMarginLoss	Creates a criterion that optimizes a multi-class classification hinge loss (margin-based loss)
	torch.nn.TripletMarginLoss	Creates a criterion that measures the triplet loss given an input tensors
	torch.nn.TripletMarginWithDistanceLoss	Creates a criterion that measures the triplet loss given input tensors

Existing layers - Utilities

● = relevant

In this category you will find a lot of utility functions…
A lot!

●	torch.nn.Flatten	Flattens a contiguous range of dims into a tensor.
●	torch.nn.Unflatten	Unflattens a tensor dim expanding it to a desired shape.

Quantization

The probability that you need it is low but I listed it here because we are working on it.
And if I need to find the link…

I will skip the following layer types (because you will not care about them):
Sparse Layers
Vision Layers
Shuffle Layers
DataParallel Layers (multi-GPU, distributed)

Leaving the PyTorch zoo

Our local AI says:

It was fun exploring the world of PyTorch with you! As we leave the zoo, I hope you had a wild time learning about the different stops, from **torch.nn.Module** to various layers and modules.
Thanks for chatting with me! If you have any more questions or want to explore more topics, feel free to ask me anytime. Have a great day!

Not so fast... Here the story really starts for us!