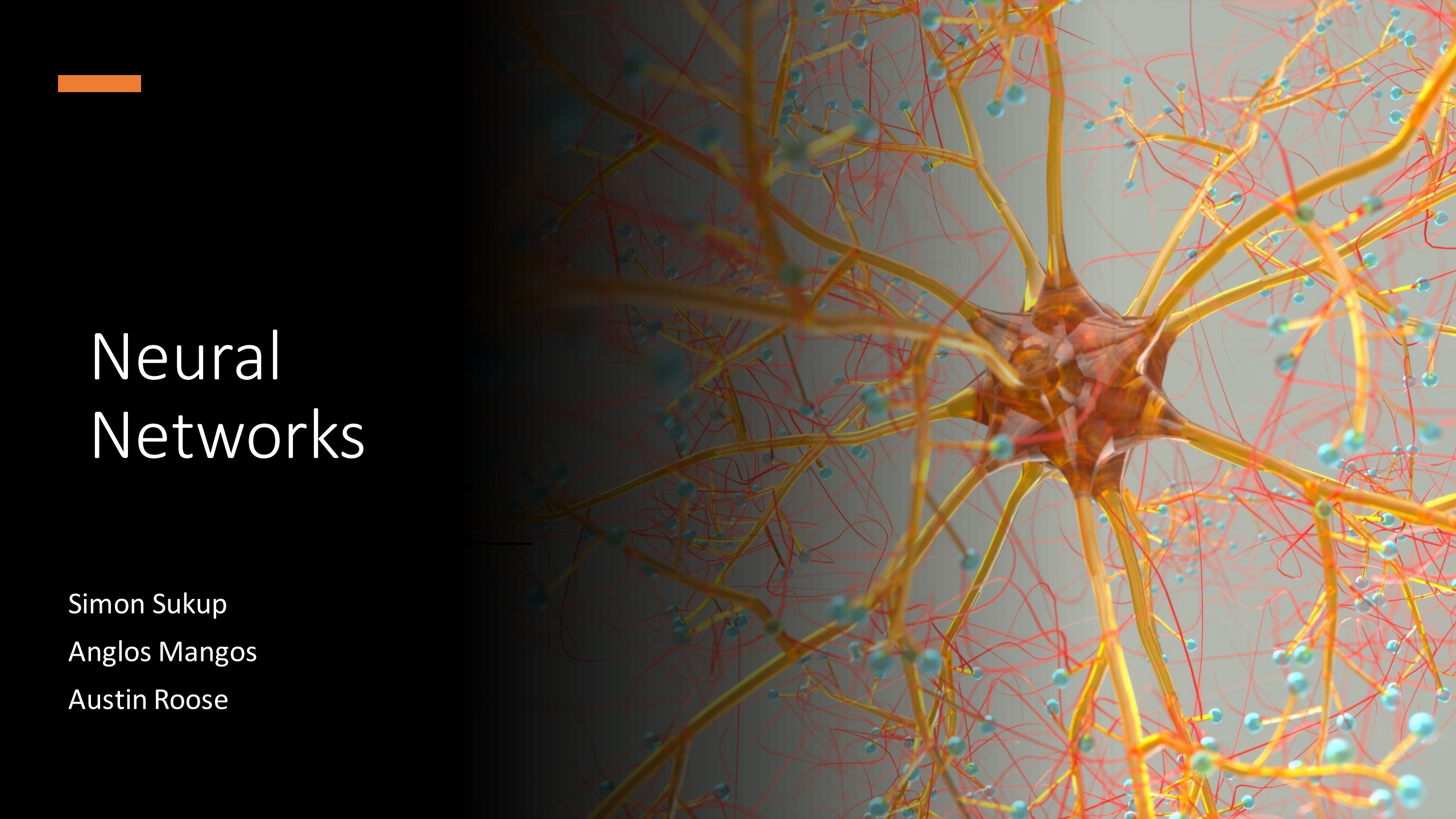
Neural Networks

Simon Sukup

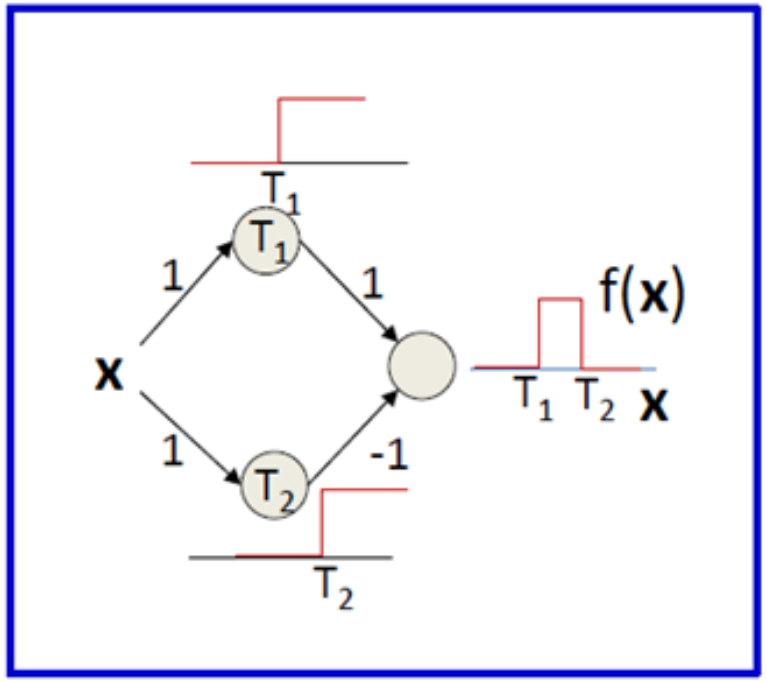
Anglos Mangos

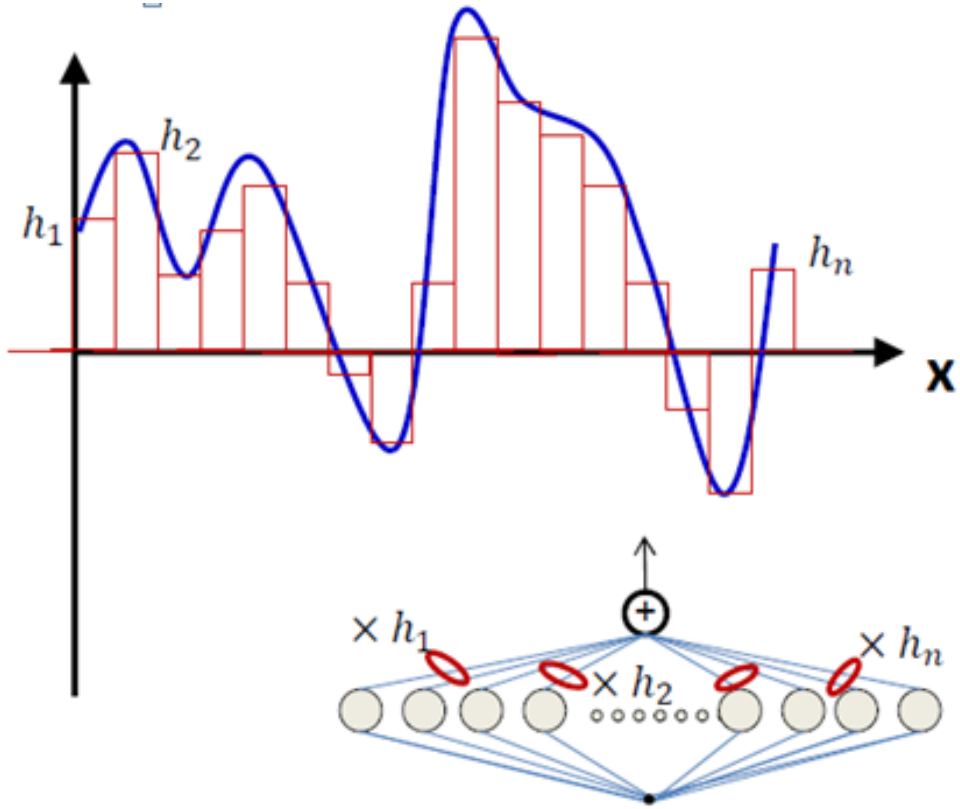
Austin Roose



Neural nets basics

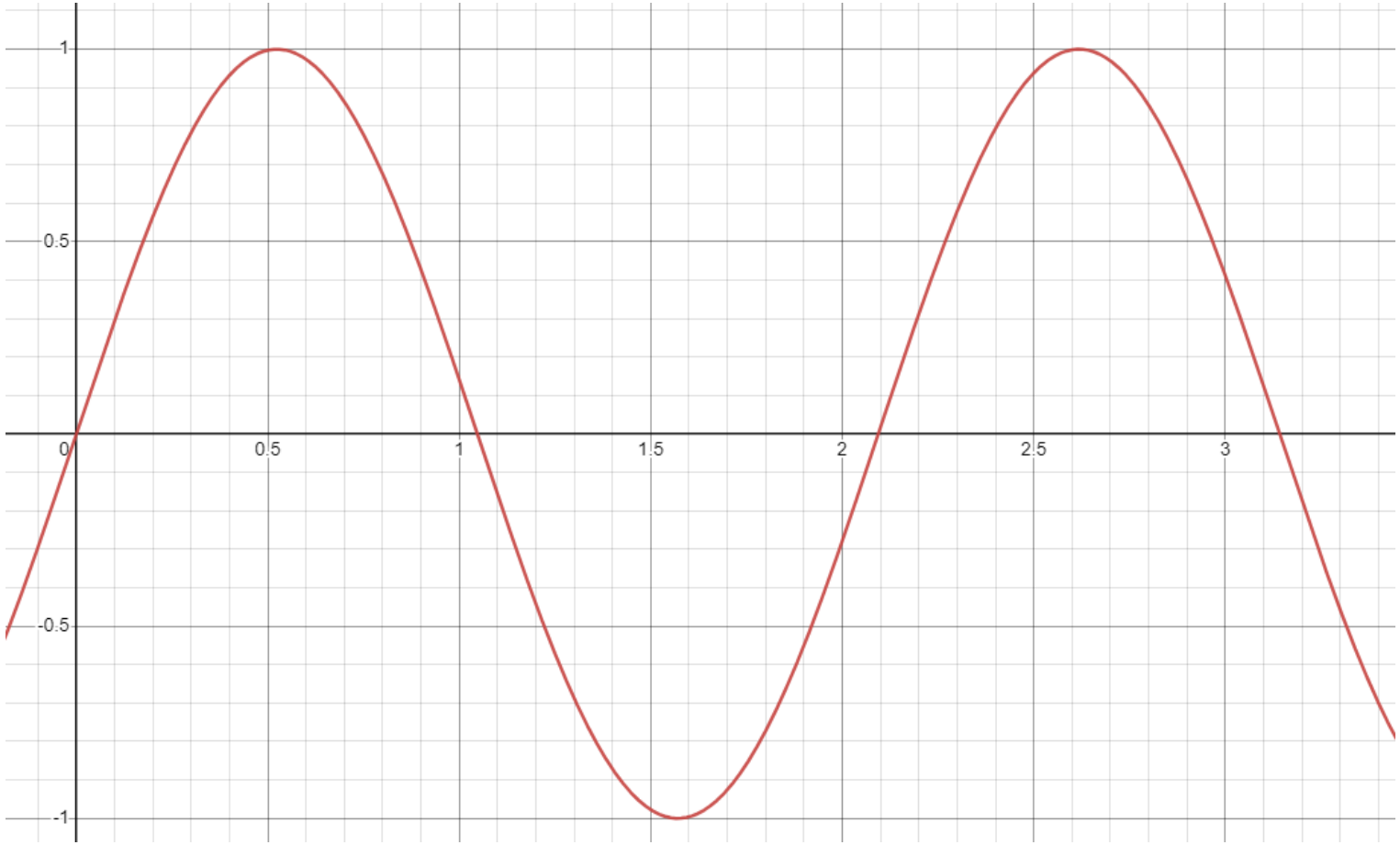
- A neural net is a universal function approximator
- This means it can approximate any function





In pytorch

Let's create a neural net which tries to approximate the sin function f=sin(3x)



The model

- This neural net should be able to approximate sin with the **correct** loss function
- This is the basic format all your neural nets will follow

```
class Net(torch.nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # A linear layer applies a linear transformation to the incoming data from 1 feature (input) to 64 features
        self.hidden1 = torch.nn.Linear(1, 64)  # 1 input, 64 hidden neurons in layer 1
        self.hidden2 = torch.nn.Linear(64, 128)  # 64 input, 128 hidden neurons in layer 2
        self.output = torch.nn.Linear(128, 1)  # 128 input, 1 output

    def forward(self, x):
        # relu is an activation function
        # using it, the neuron is activated if the input is greater than 0
        # it is used to introduce non-linearity in the network and make the network capable to learn
        # and perform more complex tasks
        x = torch.relu(self.hidden1(x))
        x = torch.relu(self.hidden2(x))
        x = self.output(x)
        return x
```

Loss function

- X is in the range -1<=x<=1
- The loss function is the criterion which the optimizer and training loop tries to minimize. For sin we will choose the mean squared error.
- We usually want to train the network until the loss converges

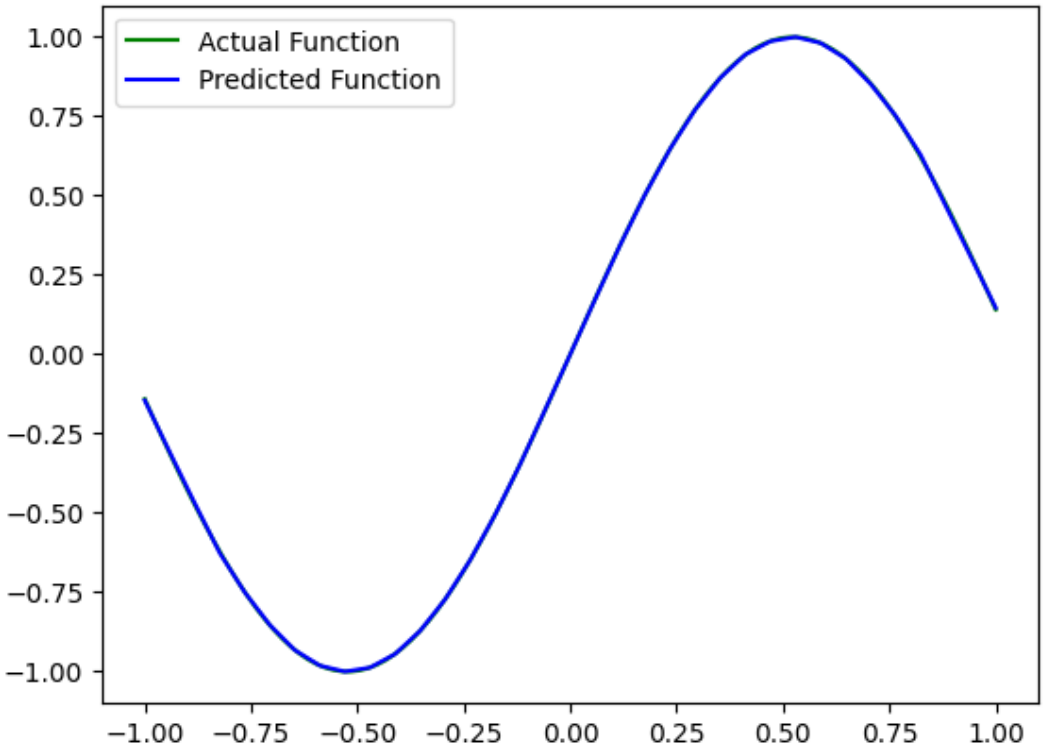
```
net = Net()  # our neural network which tries to learn the function f(x) = sin(3x)
criterion = torch.nn.MSELoss() # mean-squared error loss
optimizer = torch.optim.Adam(net.parameters(), lr=0.01)  # Using Adam optimizer

for epoch in range(1000):
    running_loss = 0.0
    optimizer.zero_grad()  # zero the gradient buffers
    outputs = net(x.unsqueeze(1))  # calculate the output from the net (net(x) approximates f(x))
    loss = criterion(outputs.squeeze(), f(x))  # f(x) is the target function (sin(3x))
    loss.backward()  # back-propagate the loss
    optimizer.step()  # Does the update, update the weights of the network
    running_loss += loss.item()  # keep track of the loss for logging purposes

    if epoch % 100 == 0:
        print("Epoch {}: Loss = {}".format(epoch, loss.detach().numpy()))
```

Output

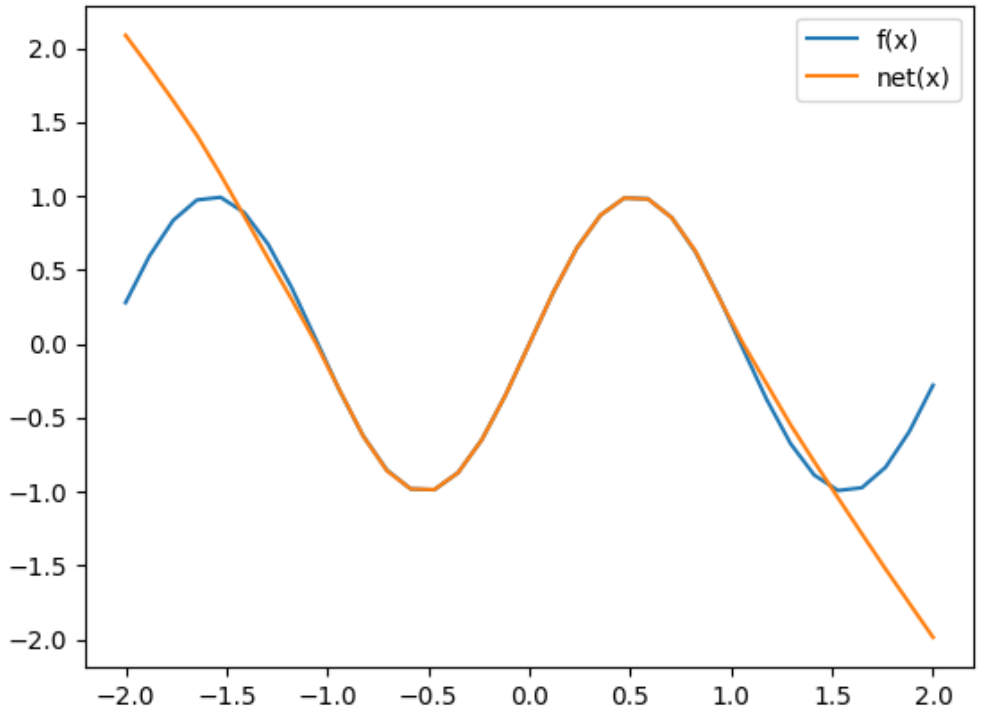
- It looks like our net was able to perfectly approximate the sin function




```
Epoch 0: Loss = 0.6337035298347473
Epoch 100: Loss = 0.00019081072241533548
Epoch 200: Loss = 5.5505668569821864e-05
Epoch 300: Loss = 0.00026257566059939563
Epoch 400: Loss = 2.8677864065684844e-06
Epoch 500: Loss = 0.0012187211541458964
Epoch 600: Loss = 3.073420884902589e-06
Epoch 700: Loss = 2.033157443293021e-06
Epoch 800: Loss = 1.5546232816632255e-06
Epoch 900: Loss = 2.1308816940290853e-05
```

Outside -1<=x<=1

- Clearly the net fails to generalize the function! It only approximated for the -1<=x<=1 range.
- This is something to remember when training and a reason why you should always have different training and testing data



Backpropagation

1.Forward Pass:

- Input data is passed through the network layer by layer from the input layer to the output layer.
- The network produces a prediction based on the current weights.

2.Compute the Loss:

- The difference between the predicted output and the actual target values (often called the "ground truth") is calculated using a loss function (e.g., mean squared error for regression tasks or cross-entropy for classification tasks).

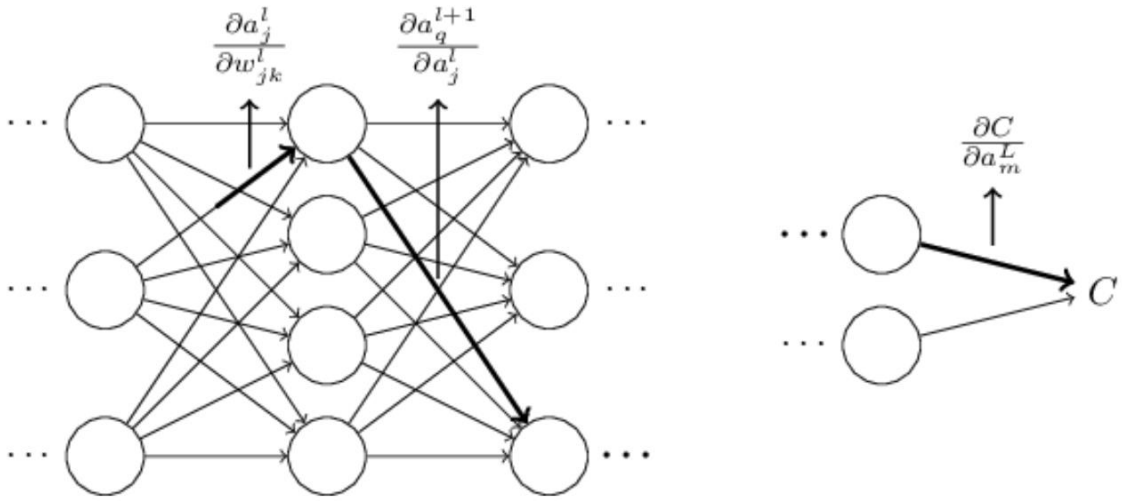
3.Backward Pass (Core of Backpropagation):

- The gradient of the loss with respect to each weight in the network is computed. This is done by applying the chain rule of calculus in a recursive manner from the output layer back to the input layer.

4.Update the Weights:

- The weights of the network are adjusted in the direction that reduces the error. This is typically done using optimization algorithms like Gradient Descent The weights are updated in the direction that reduces the loss, and this process is repeated for many iterations or epochs until the network converges.

$$\frac{\partial C}{\partial w_{jk}^l} = \sum_{mnp\ldots q} \frac{\partial C}{\partial a_m^L} \frac{\partial a_m^L}{\partial a_n^{L-1}} \frac{\partial a_n^{L-1}}{\partial a_p^{L-2}} \cdots \frac{\partial a_q^{l+1}}{\partial a_j^l} \frac{\partial a_j^l}{\partial w_{jk}^l}.$$



Backpropagation in multy-layer neural network

Optimization

- Problem: using gradient descent method some gradient vectors are long
- Use optimization for calculating gradient loss:

- **Stochastic Gradient Descent (SGD)** $g = \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)})$

$$\theta = \theta - \epsilon_k \times g$$

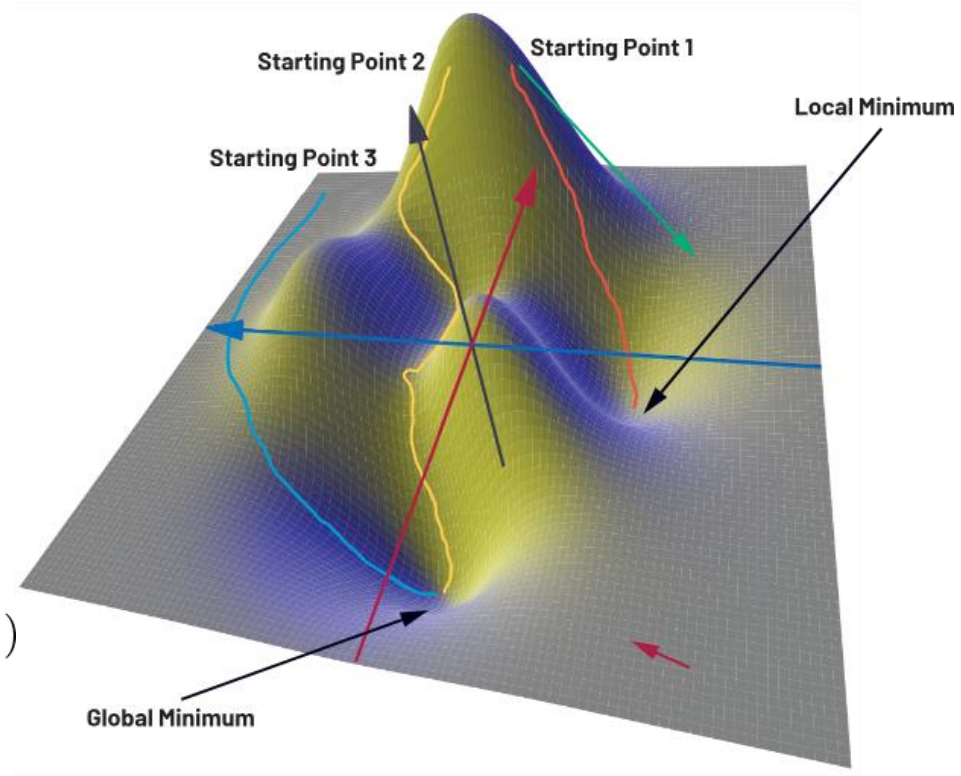
- **Momentum** $v = \alpha v - \epsilon\nabla_\theta\left(\frac{1}{m}\sum_i L(f(x^{(i)};\theta), y^{(i)})\right)$

$$\theta = \theta + v$$

- **AdaGrad** $g = \frac{1}{m}\nabla_\theta \sum_i L(f(x^{(i)};\theta), y^{(i)})$

$$s = s + g^T g$$

$$\theta = \theta - \epsilon_k \times g/\sqrt{s + eps}$$



Stochastic Gradient Descent (SGD)

- Estimate gradient from smaller sample size, use one example per iteration
- Noisy
- Mini-batch SGD
- If learning rates are optimal, converges to global minimum

$$w := w - \eta \nabla Q_i(w).$$

Momentum

- Keep direction and magnitude from previously updated gradient descent
- Contribution defined by decay rate

```
# Recalculating the difference
grad = np.array(gradient(x_batch, y_batch, vector), dtype_)
diff = decay_rate * diff - learn_rate * grad
```

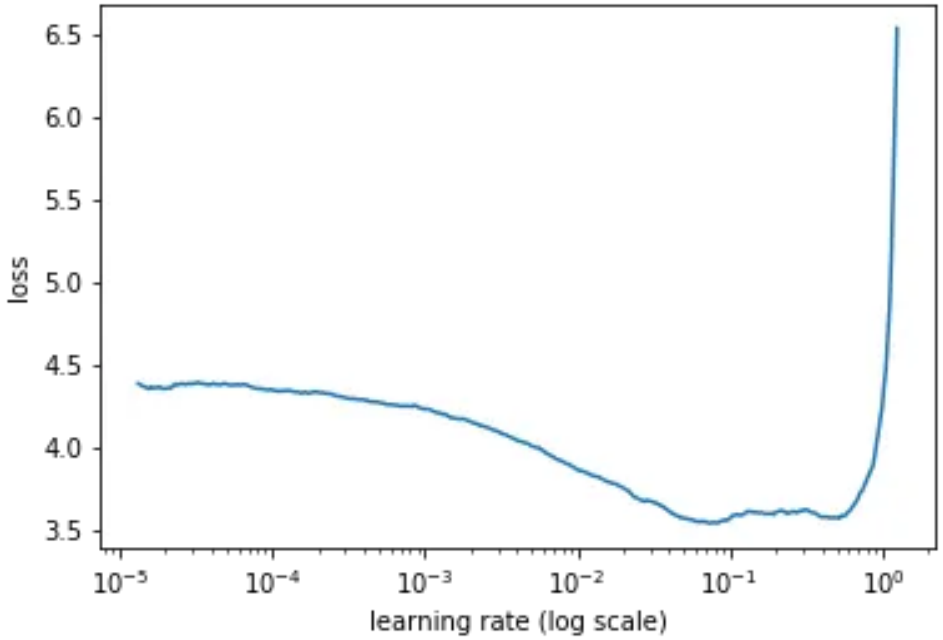
```
sgd = tf.keras.optimizers.SGD(learning_rate=0.1, momentum=0.9)
```

Adaptive Gradient Algorithm (Adagrad)

- Component-wise updates to parameters
- Efficient for parse data

Learning rate

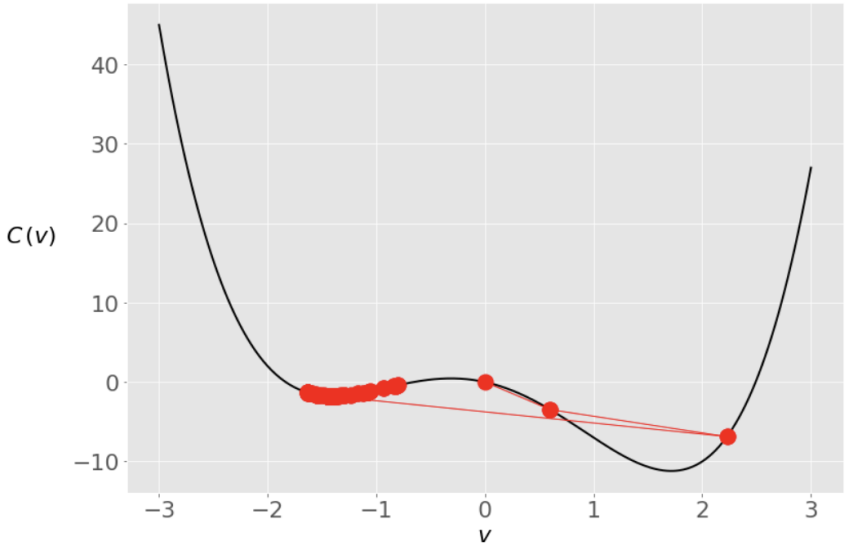
- How much θ follows gradient estimate g
- Not too high – overshooting optimal
- Not too low – will not converge
- Optimal learning rate – increase learning rate exponentially for batches, record loss for each rate, get point where loss decreases fastest
- Learning rate finder by fast.ai
- Efficient way for changing learning rate - cyclic learning rate (CLR) methods



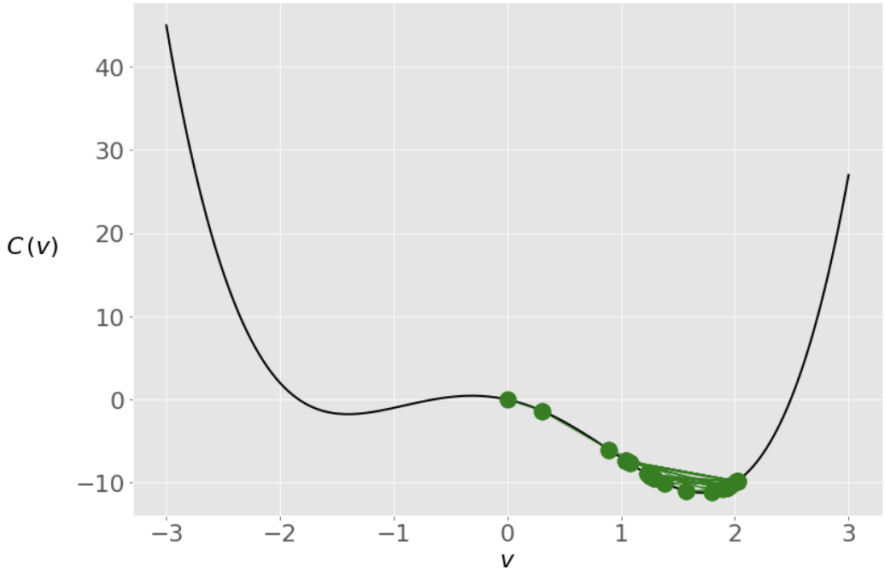
Different rates

$v^4 - 5v^2 - 3v. \longrightarrow 4v^3 - 10v - 3.$

```
gradient_descent(
    gradient=lambda v: 4 * v**3 - 10 * v - 3, start=0,
    learn_rate=0.2
)
```




```
gradient_descent(
    gradient=lambda v: 4 * v**3 - 10 * v - 3, start=0,
    learn_rate=0.1
)
```



Activation functions

You must choose the activation function for your output layer based on the type of prediction problem that you are solving.
The activations in hidden layer re often given by architecture. Follow structure of standard architectures.

ReLU (Rectified Linear Unit)

- **When to Use**: First choice for hidden layers in feedforward neural networks and convolutional neural networks.
- **Advantages**: Helps mitigate the vanishing gradient problem. Computationally efficient.
- **Caveats**: Can suffer from the "dying ReLU" problem, where neurons can sometimes get stuck and stop updating.

Tanh (Hyperbolic Tangent)

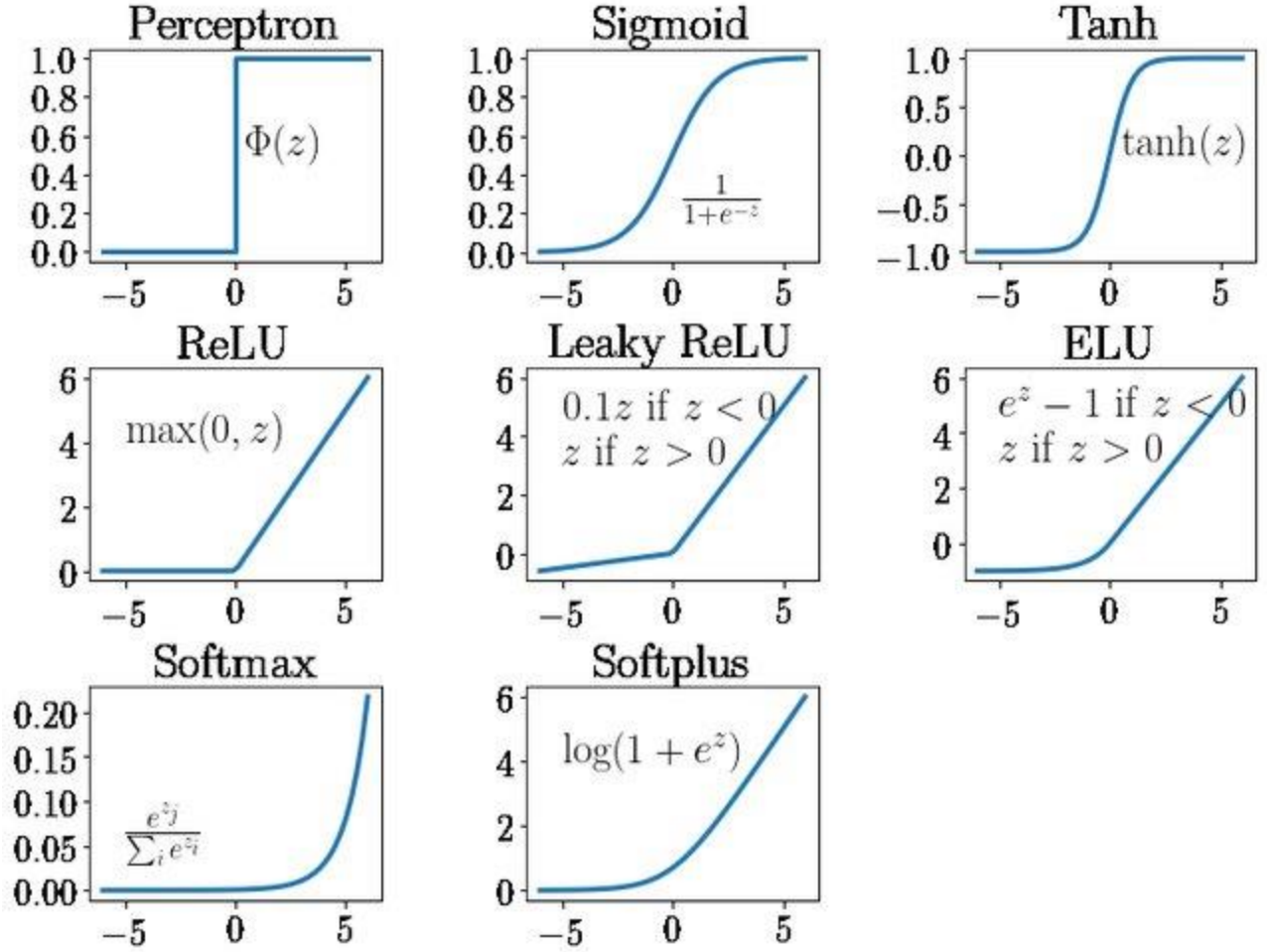
- **When to Use**: Hidden layers in feedforward networks. It's like a scaled sigmoid function.
- **Advantages**: Maps input values to the (-1, 1) range.
- **Caveats**: Can still suffer from the vanishing gradient problem, though less severely than the sigmoid.

Sigmoid

- **When to Use:** Historically used for hidden layers in feedforward networks and output layers in binary classification problems.
- **Advantages:** Maps input values to the (0, 1) range.
- **Caveats:** Can cause vanishing gradient problem, especially in deep networks.

Softmax

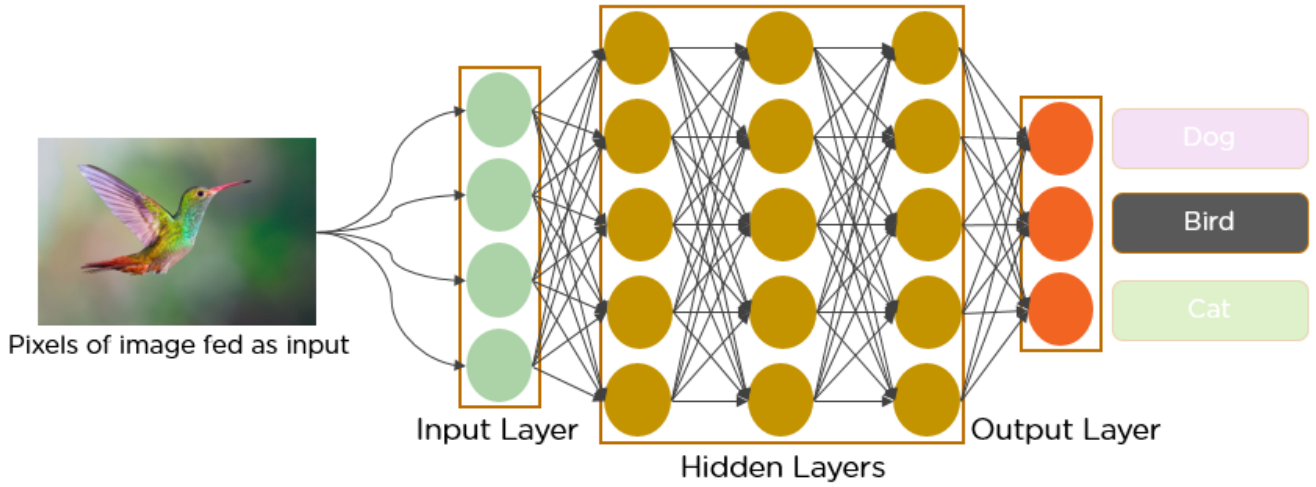
- **When to Use**: Output layer for multi-class classification problems.
- **Advantages**: Converts a vector of numbers into a probability distribution.
- **Caveats**: Only used in the output layer due to its specific purpose.



Why are simple networks inefficient for CV?

- Consider a 200 x 200 x 3 image
  - This is an RGB image with height and width 200 pixels
  - We can represent this by a 200x200x3 = 120,000 element vector

- How many parameters do we need for an MLP with one fully connected hidden layer of 10 units?
- 200 * 200 * 3 * 10 + 10 = 1,200,010 !!!
- **Our network would be huge and nearly impossible to train.**

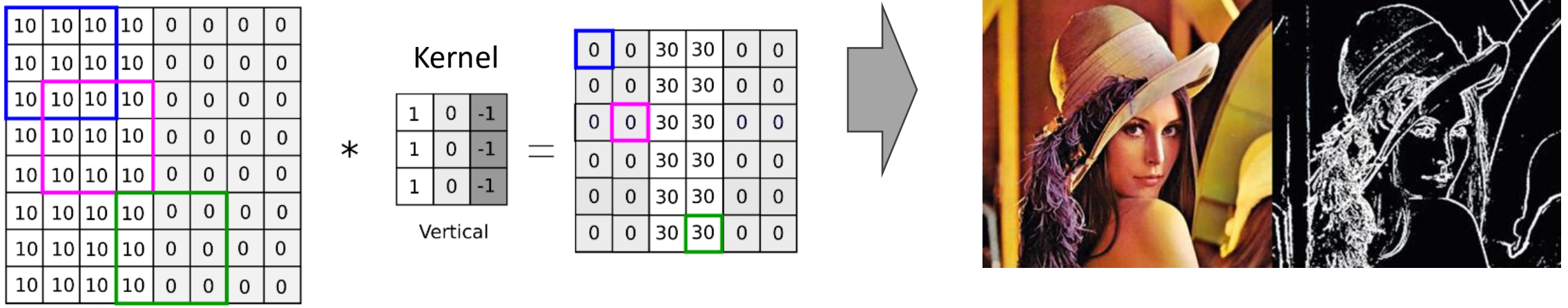


An inefficient but possible technique for image classification. Source: [1]

Digital filters and convolution

Convolution steps

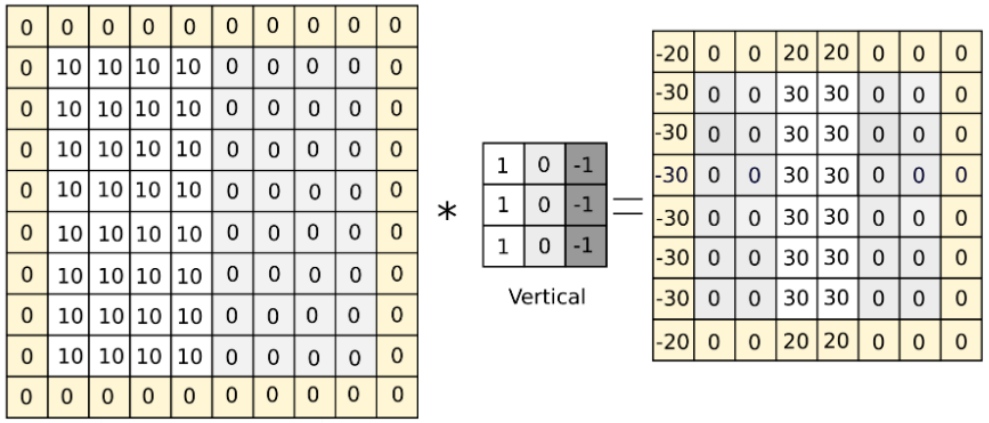
1. Overlaying the filter on top of the image at some location.
2. Performing **element-wise multiplication** between the values in the filter and their corresponding values in the image.
3. Summing up all the element-wise products. This sum is the output value for the **destination pixel** in the output image.
4. Repeating for all locations.



Use of kernels for digital image processing via convolution. Source: [2]

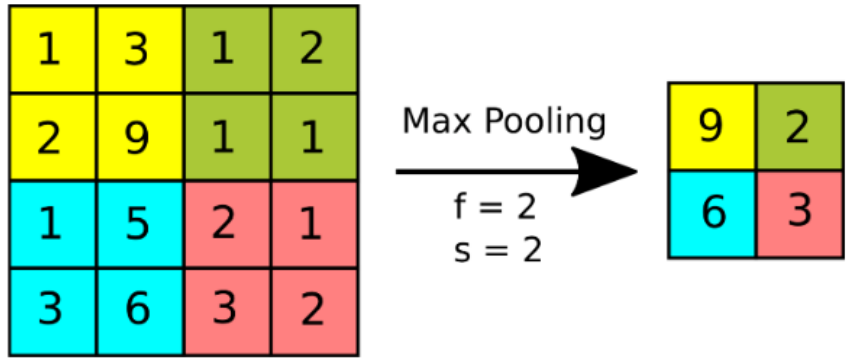
Final image after edge detection.

Padding



Used to keep the spatial dimensions (width and height) of the output the same as the input, especially when using kernels/filters that would otherwise reduce the dimensions. Source: [2]

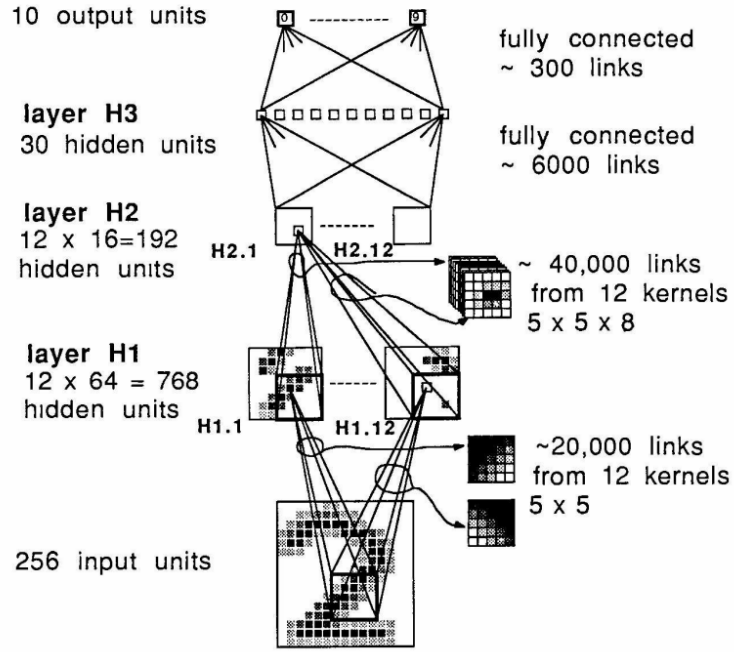
Maxpooling and stride



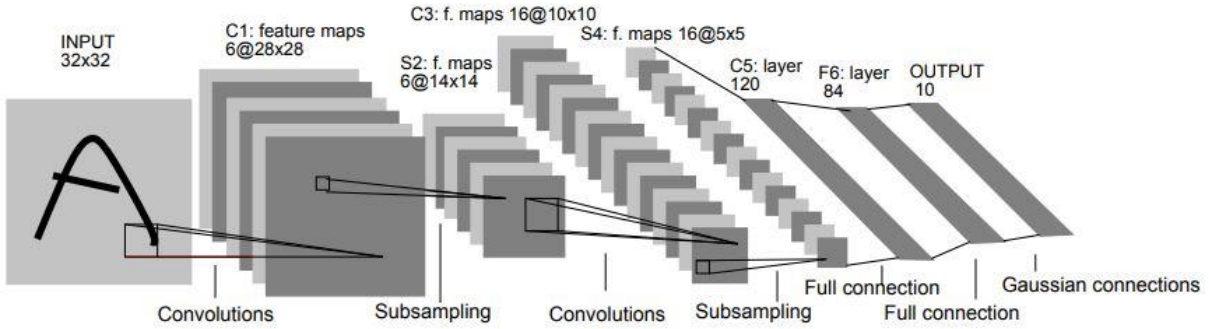
Maxpooling is an operation where we slide a window (often 2x2) over an image and select the maximum value from the window. This operation reduces the spatial dimensions of the image.
stride is number of pixels by which we slide our filter/kernel or pooling over the input Source: [2]

History

- Neural networks extract features, how does human extract features from image? By looking at each part of image separately as well as surroundings
  - we have to look at features from low level to high level
- We can consider standard digital image processing techniques to achieve this hierarchical feature extraction
- In the late 1980s and early 1990s, Yann LeCun and his collaborators introduced the modern architecture of CNNs and applied backpropagation for supervised learning in these networks.
- Yann LeCun et al developed "LeNet-5" in 1998, which was designed for handwritten and machine-printed character recognition.



The architecture of LeNet-5 from original paper. Source: [3]



Alternative visualization of the LeNet5 architecture. Source: [4]

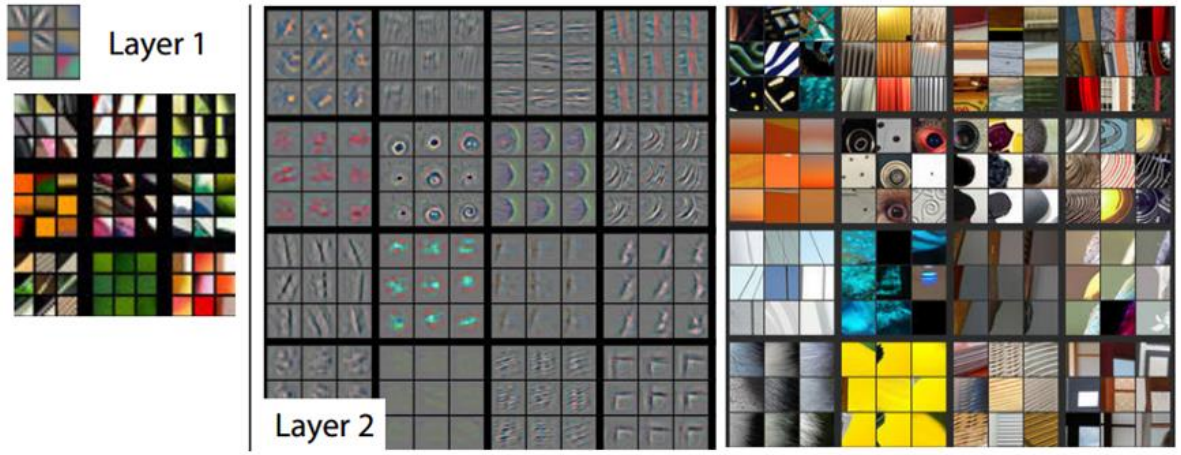
Feature maps and hierarchy

Lower Layers (Early Layers):

- **Basic Features**: These layers detect low-level features such as edges, colors, and textures. The filters/kernels in these layers often look like edge detectors or color blobs.
- **Granularity**: The receptive field (portion of the input image seen by the filter) at this stage is small, so the features detected are simple and local.

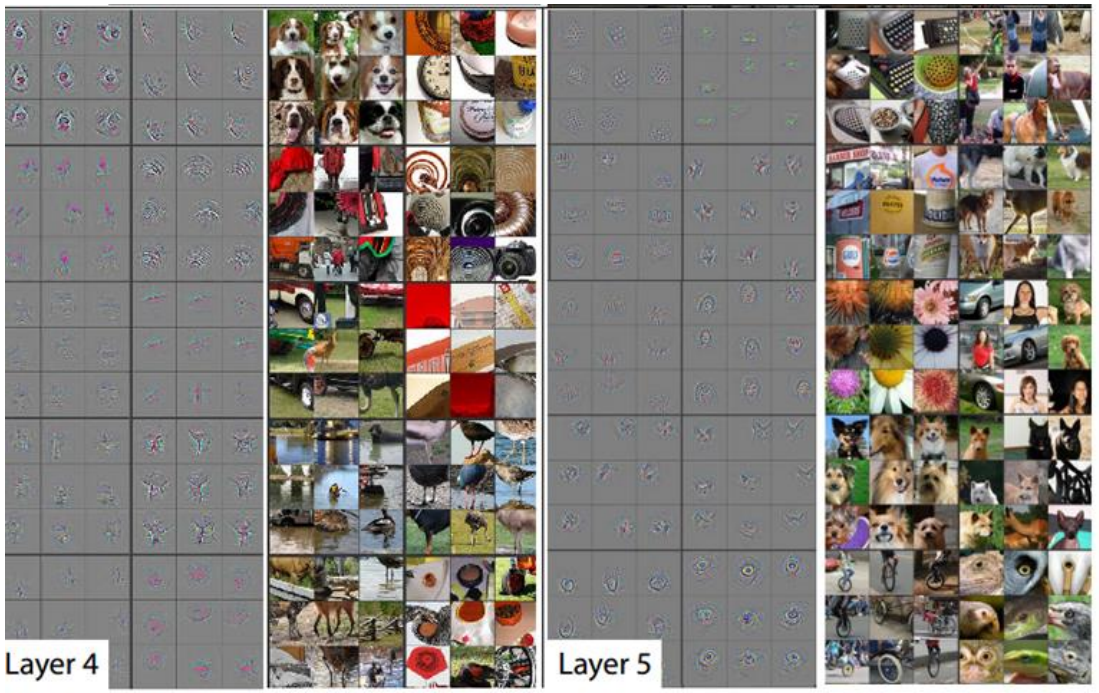
Higher Layers (Deeper Layers):

- **Abstract and High-Level Features**: These layers capture more abstract information. The patterns recognized here might correspond to parts of objects or even entire objects in the case of images (e.g., a dog's ear, a car wheel, or a bird's wing).
- **Context**: Because these layers have aggregated information from many previous layers, they understand the context from larger portions of the input image.



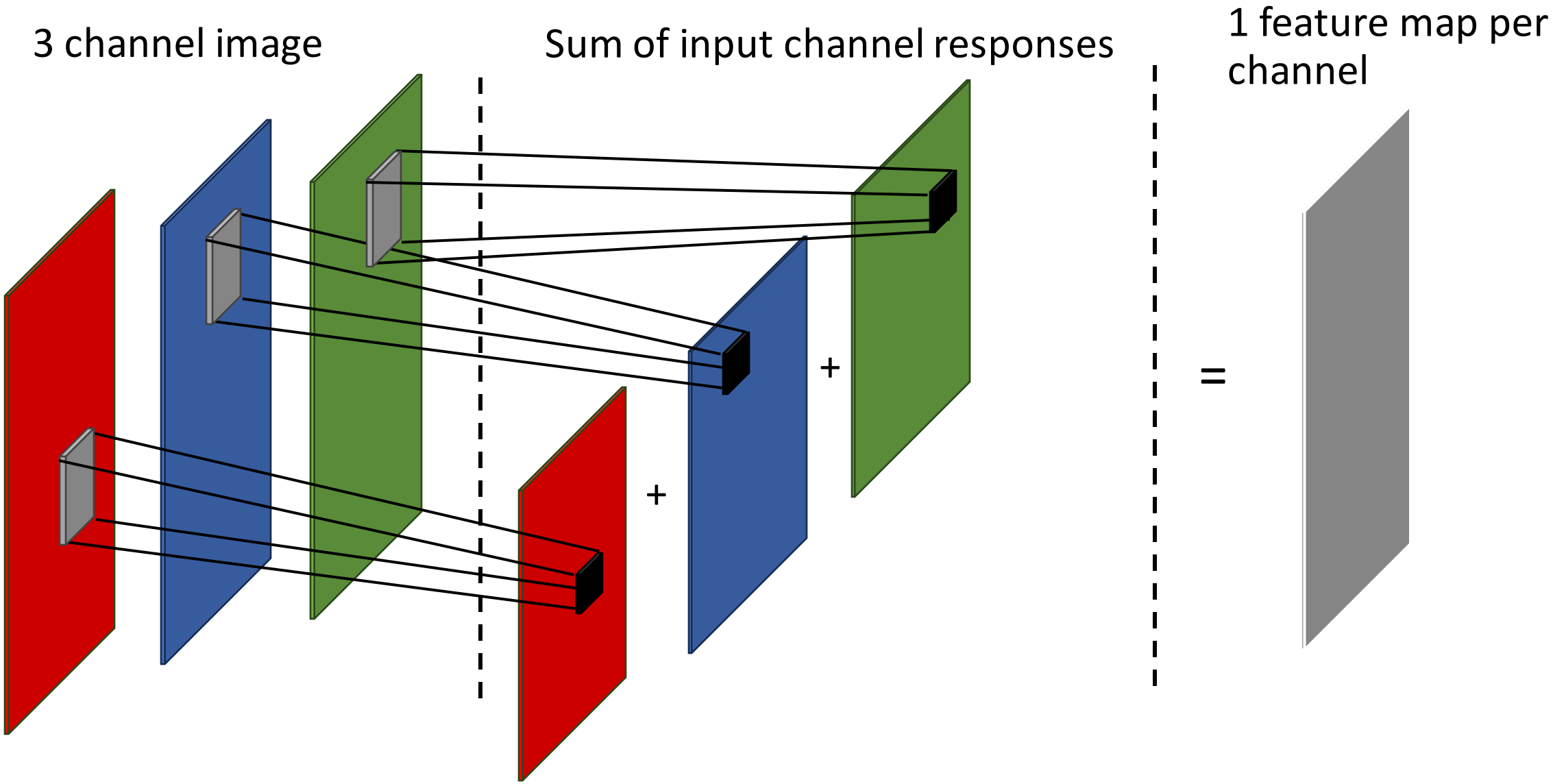
Visualizations of Layer 1 and 2. Each layer illustrates 2 pictures, one which shows the filters themselves and one that shows what part of the image are most strongly activated by the given filter. For example, in the space labled Layer 2, we have representations of the 16 different filters (on the left)

Source: [1]



isualizations of Layers 3, 4, and 5

Convolution in practice per output channel (kernel)



Convolution in practice

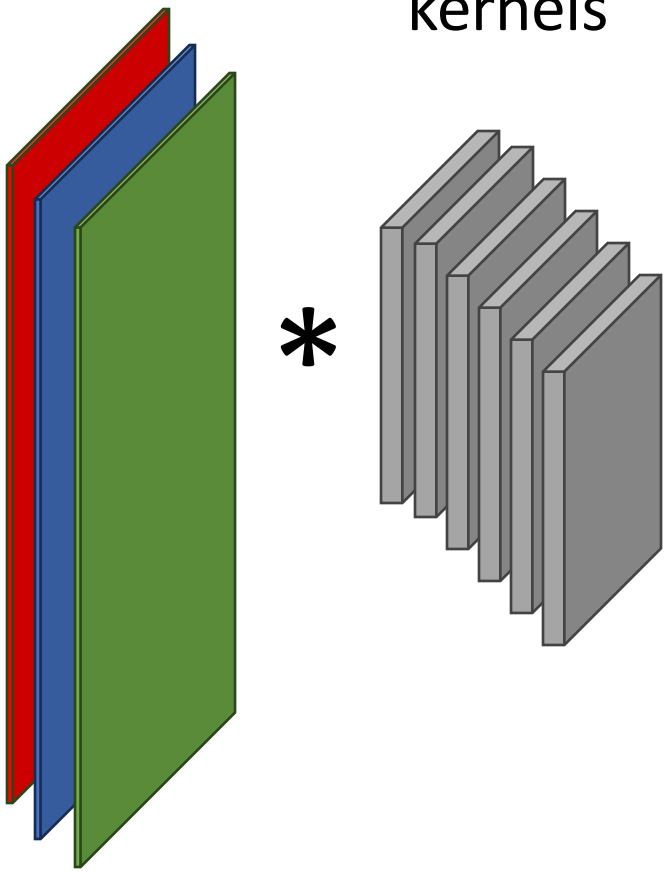
CONV2D

CLASS torch.nn.Conv2d(*in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None*) [SOURCE]

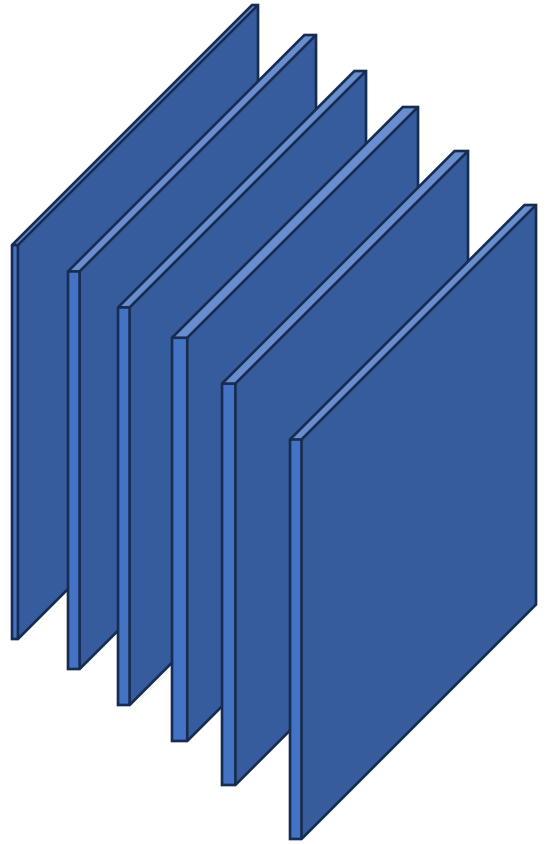
3 channel image

kernels

Feature maps

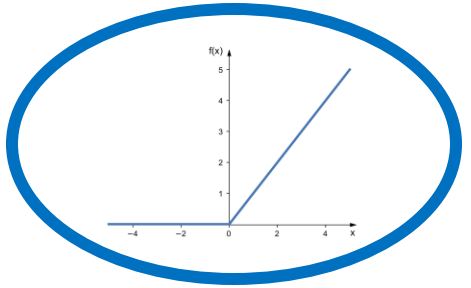


In_channels=3



out_channels=6

Activation function

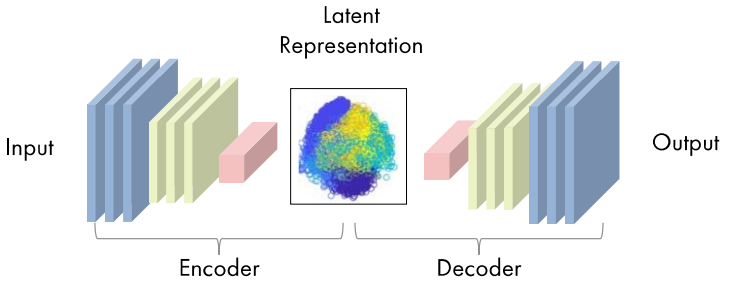


History of architectures

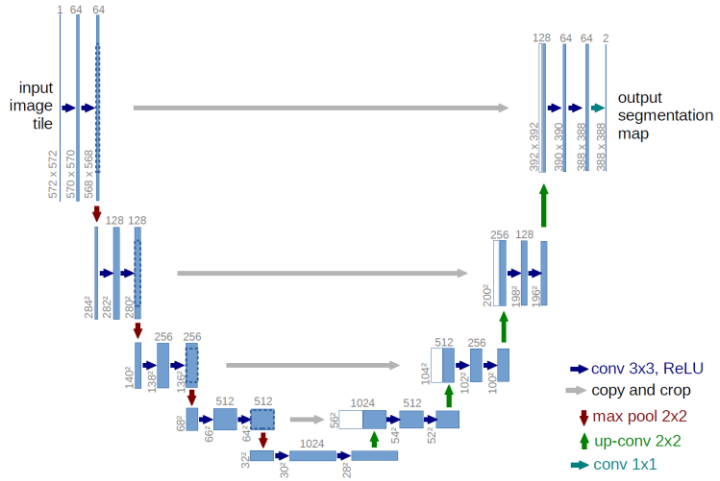
- **LeNet-5 (1998)**:
  - Developed by Yann LeCun.
  - One of the earliest CNNs designed for handwritten digit recognition.
  - Consists of two convolutional layers followed by pooling layers and then fully connected layers.
- **AlexNet (2012)**:
  - Developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton.
  - Won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012.
  - Introduced the use of the ReLU activation function and dropout.
- **VGG (2014)**:
  - Developed by the Visual Geometry Group at Oxford.
  - Known for its simplicity and depth, with configurations having 16 (VGG16) or 19 (VGG19) layers.
  - Uses small 3x3 convolutional filters throughout.
- **GoogLeNet/Inception (2014)**:
  - Developed by researchers at Google.
  - Introduced the "Inception module" which allows the network to choose from different convolutional filter sizes.
  - Won the ILSVRC in 2014.
- **ResNet (2015)**:
  - **Developed by Kaiming He and colleagues at Microsoft Research.**
  - **Introduced "residual connections" or "skip connections" that allow gradients to flow through the network, enabling the training of very deep networks (e.g., 152 layers).**
  - **Won the ILSVRC in 2015.**
- **DenseNet (2017)**:
  - Extends the idea of ResNet by connecting each layer to every other layer in a feed-forward fashion.
  - Ensures maximum information flow between layers in the network.
- **MobileNet (2017)**:
  - Developed by Google researchers.
  - Designed for mobile and embedded vision applications.
  - Uses depthwise separable convolutions to reduce the number of parameters and computational cost.
- **EfficientNet (2019):**
  - **Also developed by Google researchers.**
  - **Uses a compound scaling method to scale up the network in depth, width, and resolution.**
  - **Achieves state-of-the-art accuracy with significantly fewer parameters.**
- **Vision Transformers (ViT) (2020)**:
  - While not a traditional CNN, ViTs have become popular for vision tasks.
  - Divides the image into fixed-size patches, linearly embeds them, and then processes them with Transformer blocks.
  - Achieves competitive results with CNNs on image classification tasks.

Useful model types for specific application

- **Autoencoders:**
  - Purpose: Autoencoders are designed for unsupervised learning tasks, primarily for dimensionality reduction and anomaly detection. They learn to encode input data into a compressed representation and then decode it back to the original form.
  - Distinct Feature: The bottleneck in the middle, which forces the network to learn a compressed representation of the input data
  - **U-Net**: designed for image segmentation tasks, where the goal is to classify each pixel in an image into a specific category. Originally developed for biomedical image segmentation but it is basis of many projects outside biomedical sphere.
  - **Variational autoencoders:** used for generative modeling and unsupervised learning.
- **Generative Adversarial Networks (GANs)**:
  - **Purpose**: Data generation.
  - **Description**: Consists of a generator that produces data and a discriminator that distinguishes between real and generated data. They are trained adversarially.
- **FPN based models for segmentation:**
  - Feature Pyramid Networks (FPN) based models for segmentation leverage a multi-scale feature hierarchy to detect and segment objects of varying sizes in images.
  - Panoptic FCN, PanopticFormer, Transformer based visual segmentation networks
- **Diffusion Models**:
  - class of generative models that transform a simple noise distribution into a complex data distribution through a series of diffusion (or denoising)
  - DDPM, VDM...



Autoencoder



U-Net

```python
import torch.nn as nn
class SimpleCNN(nn.Module):
    def __init__(self, input_channels, num_classes):
        super(SimpleCNN, self).__init__()
        # First sequence: CONV -> BatchNorm -> ReLU -> Dropout
        self.conv1 = nn.Conv2d(in_channels=input_channels, out_channels=32, kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(32)
        self.relu1 = nn.ReLU()
        self.dropout1 = nn.Dropout(0.5)
        # Second sequence: CONV
        self.conv2
        # Fully Cor
        self.fc = r
    def forward(sel
        # First sec
        x = self.cc
        x = self.br
        x = self.re
        x = self.dr
        # Second se
        x = self.cc
        # Flatten and pass through FC
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

# Instantiate the model
```

BATCHNORM2D

CLASS torch.nn.BatchNorm2d(*num_features*, *eps=1e-05*, *momentum=0.1*, *affine=True*, *track_running_stats=True*, *device=None*, *dtype=None*) [SOURCE]

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 .

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Pitfalls of deep learning CV models

- **Vanishing Gradient**:
  - **Problem**: As the network becomes deeper, gradients during backpropagation can become very small, causing the network to stop learning.
  - **Solution**:
    - Use activation functions like ReLU (Rectified Linear Unit) that don't saturate for large input values.
    - Implement skip or residual connections (as in ResNet) to allow gradients to flow through the network.
- **Exploding Gradient**:
  - **Problem**: Gradients can become too large, causing model weights to update in extreme ways, leading to model instability.
  - **Solution**:
    - Gradient clipping: Set a threshold value and scale gradients that exceed this threshold.
    - Use gentler activation functions or architectures designed to mitigate this issue.
- **Overfitting**:
  - **Problem**: The model performs well on the training data but poorly on unseen data, indicating it has memorized the training data.
  - **Solution**:
    - Use regularization techniques like dropout, L1, or L2 regularization.
    - Augment the training data to increase its diversity.
    - Use early stopping based on validation performance.

Thank you for your attention

Citation

[1] Manav Mandal, AuthorFirstInitial. (2021). Title of photograph [Photograph]. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/

[2] Anh H. Reynolds, Convolutional Neural Networks (CNNs) URL: https://anhreynolds.com/blogs/cnn.html.

[3] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-Based Learning Applied to Document Recognition. Proceedings of the IEEE, 86(11), 2278-2324.

[4] ómez, Nuria & López, Luis & Albert, Alberto & Llamas, Javier. (2020). Image Classification with Convolutional Neural Networks Using Gulf of Maine Humpback Whale Catalog. Electronics. 9. 731. 10.3390/electronics9050731.