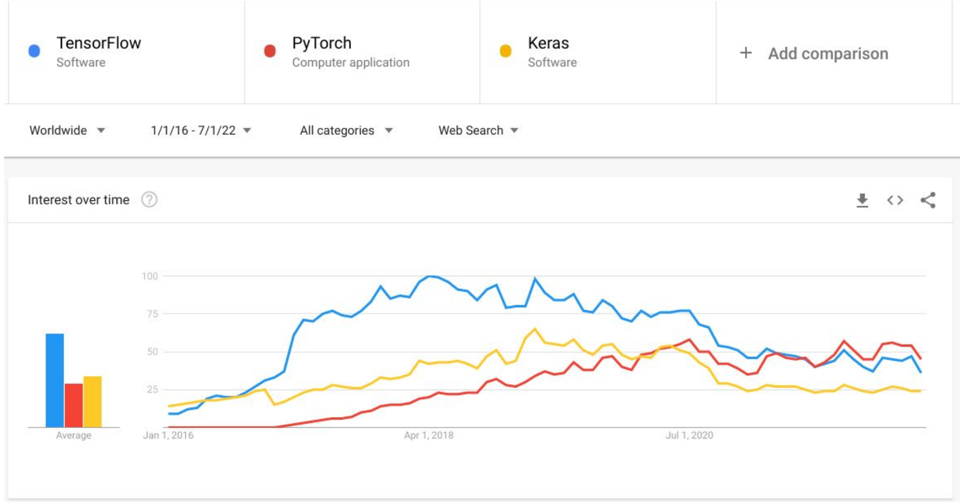
Machine Learning:
Introduction to Deep Learning, Convolutional Neural Networks

Schedule for This Part

- Introduction to Machine Learning, Decision Trees
- **Introduction to Deep Learning, Convolutional Neural Networks:**
  - **Artificial (Deep) Neural Networks**
  - **Convolutional Neural Networks**
- Unsupervised Machine Learning, Autoencoders
- Introduction to Graph Neural Networks

Frameworks

- We will work with **PyTorch**.
- Alternatively TensorFlow and Keras are a popular choice.



Brief History of Artificial Neural Networks

- 1943: McCulloch & Pitts: simple neural networks with electrical circuits

- 1958: Rosenblatt: works on **perceptron**

- 1959: Widrow & Hoff: first neural network applied to real world problem (**ADALINE**)

- 1969: Minsky & Papert: proved limitations of perceptron

- 1986: Rumelhart, Hinton & Wiliams: **backpropagation** for multi-layer perceptron

- 2012: Krizhevsky: CNN (**AlexNet**) wins image recognition competition

Artificial Neural Networks

- You’ve just learned about BDTs.

What about highly non-linear data?

Big datasets?

Data with many input features (like images)?

- We can transform the input space but we often don't know how *a priori*

Universal Approximation Theorem

A single hidden layer neural network with a linear output unit can approximate any continuous function arbitrary well, given enough hidden units

Hornik 1991

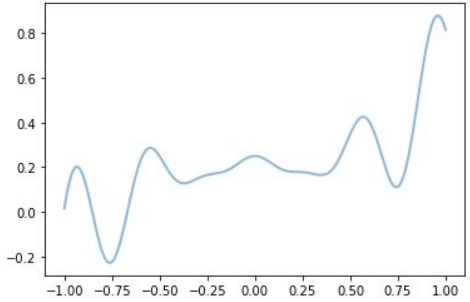
Artificial Neural Networks

Universal Approximation Theorem

A single hidden layer neural network with a linear output unit can approximate any continuous function arbitrary well, given enough hidden units

Hornik 1991

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

f = lambda x: 0.2 + 0.4*x**3 + 0.3*x*np.sin(15*x) + 0.05*np.cos(20*x)
X = np.linspace(-1,1., 1024)
y = f(X)

plt.plot(X, y, '-', alpha=0.5, lw=2);
```



Artificial Neural Networks

Universal Approximation Theorem

A single hidden layer neural network with a linear output unit can approximate any continuous function arbitrary well, given enough hidden units

Hornik 1991

```
import torch
from utils import ShallowNN

model = ShallowNN()
model.load_state_dict(torch.load('./media/universal_approximator'))
model.eval()
print(model)
```

```
ShallowNN(
  (regressor): Sequential(
    (0): Linear(in_features=1, out_features=2000, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=2000, out_features=1, bias=True)
  )
)
```

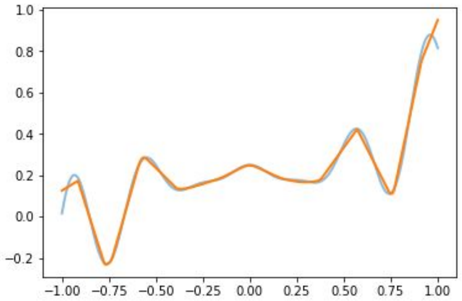
Artificial Neural Networks

Universal Approximation Theorem

A single hidden layer neural network with a linear output unit can approximate any continuous function arbitrary well, given enough hidden units

Hornik 1991

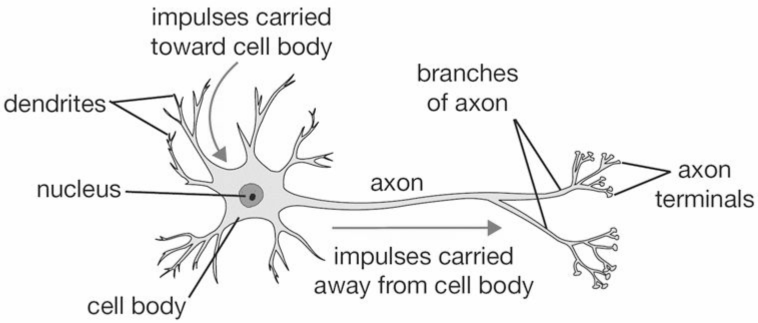
```
y_pred = model(torch.Tensor(X).unsqueeze(1))

plt.plot(X, y, '-', alpha=0.5, lw=2);
plt.plot(X, y_pred.data.squeeze().numpy(), lw=2);
```

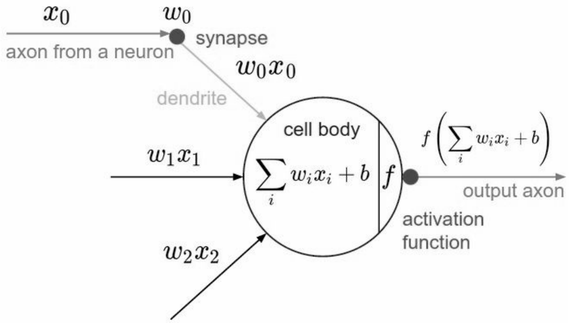


Artificial Neural Networks

- Neural networks are inspired by biological neurons.

  x: neuron (node) input;
  w: neuron weight;
  b: bias;
  f: activation function

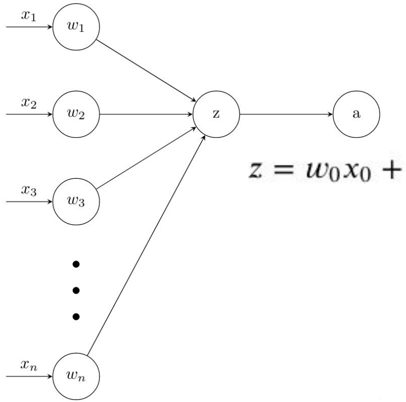




Credit

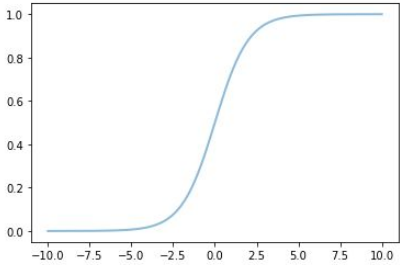
Artificial Neural Networks

- Neural networks are inspired by biological neurons.
- Affine transformation of the input data.
- Followed by (non-linear) activation, e.g. sigmoid function $\frac{1}{1+exp(-z)}$



$$z = w_0x_0 + w_1x_1 + \ldots + w_nx_n + b = \mathbf{w}^T\mathbf{x}$$
$$a = \sigma(z)$$

```
z = np.linspace(-10,10., 1024)
y = torch.sigmoid(torch.tensor(z)).numpy()

plt.plot(z, y, '-', alpha=0.5, lw=2);
```



- Nodes are combined into layers.

From Neuron to Network

- A shallow neural network, given wide enough hidden layer should approximate well a given function *f*. In practice this is quite difficult…
- Stacking more layers instead improves performance. Why?
- Space folding:

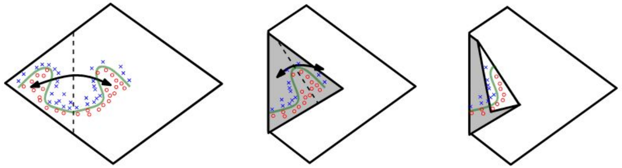
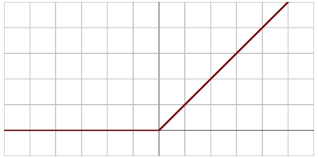


Figure 3: Space folding of 2-D space in a non-trivial way. Note how the folding can potentially identify symmetries in the boundary that it needs to learn.

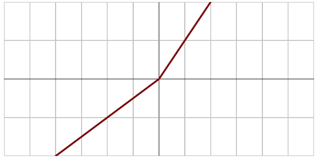
Source

More Activation Functions

- Sigmoid. Expensive, saturates for low and high output.
- ReLU. Non-negative, cheap but dies for x < 0.



- Leaky ReLU. Non-zero for negative values.

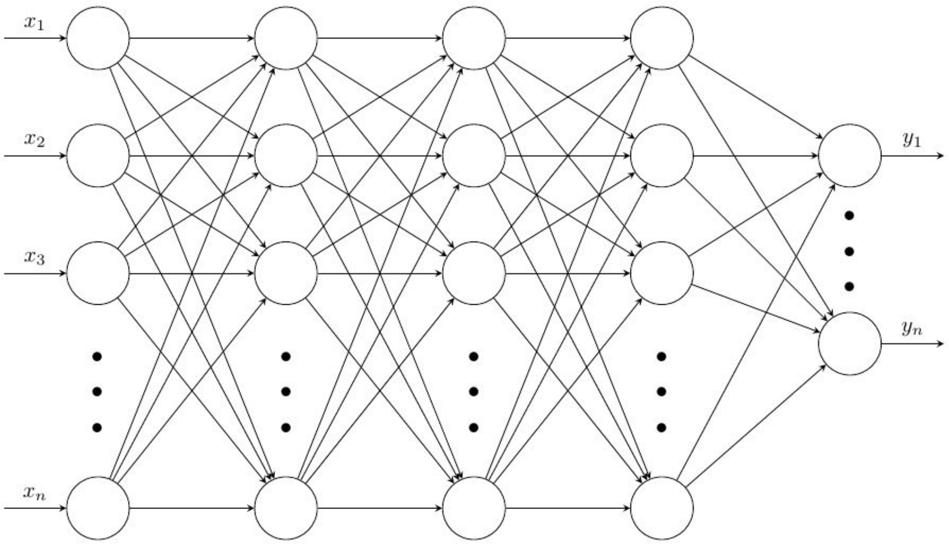


- Softmax, $\frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}}$. Produces probability over classes, i.e. use it for classification.

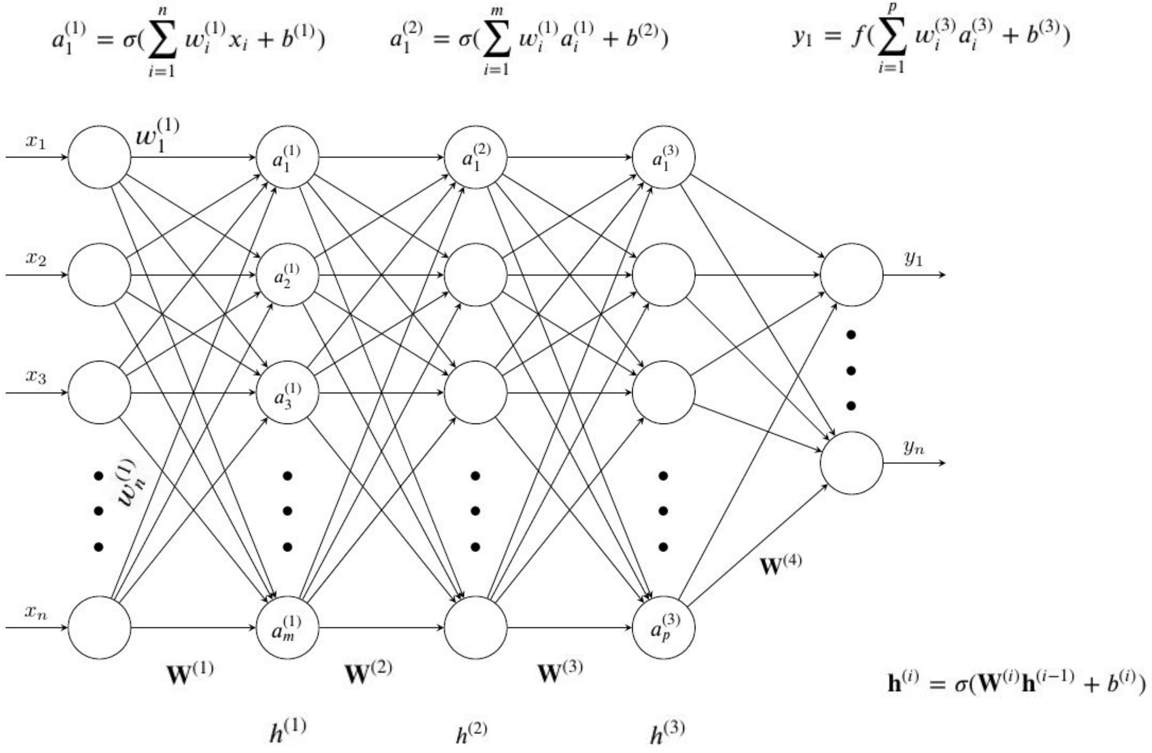
Visualizations from Wikipedia

Question Interlude

- How many hidden layers does this network have?



Inference (Feed-forward Pass)



Training Neural Network

- For the pair of input x_i and corresponding label y_i. We want to **minimize the loss function** E.
- Loss function quantifies how well the model is achieving the learning objective, e.g. mean squared error $\sum_{i=1}^{D}(\hat{y}_i - y_i)^2$ or cross-entropy loss $-(y\log(p) + (1-y)\log(1-p))$.
- Model parameters θ: weights and biases.
- X is described by a vector of variables, aka features.

Solution:

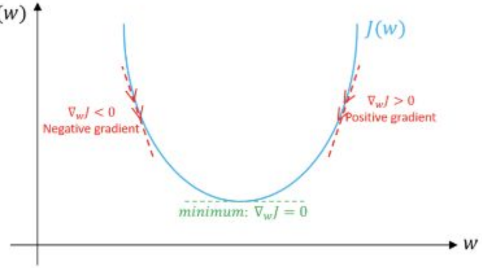
- Inefficient: Random search of θ, for which we minimize E.
- Much better: Neural networks layers are differentiable, we can use gradient descent.
- Another alternative: hebbian learning.

Gradient Descent

- We normally minimize things by evaluating the derivatives (direction towards minimum).
- Gradient descent computes the gradient of the cost function w.r.t. to the current parameters θ for the entire training dataset.
- At each training step k update model parameters to move towards steepest decline:

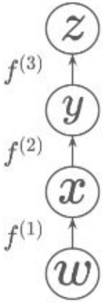
$$\theta_{k+1} \leftarrow \theta_k - \eta \cdot \nabla \theta_k J(\theta_k),$$ where $J(\theta_k)$ is the Jacobian and η is the step size

- Adjustment step is determined by learning rate η (hyperparameter).



Chain Rule

- Computing derivatives of a *base* function: decompose composite function into a set of base ones and differentiate them one by one.



$$
\begin{aligned}
&\frac{\partial z}{\partial w} \\
&= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\
&= f^{(3)\prime}(y) f^{(2)\prime}(x) f^{(1)\prime}(w)
\end{aligned}
$$

Backpropagation

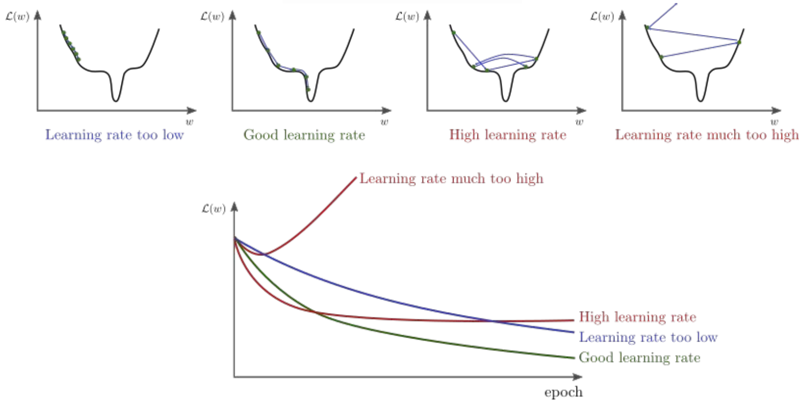
Backward propagation of errors

- Make forward pass through the network to calculate the output and the corresponding loss.
- Do a backward pass go back through the network to calculate gradients for all weights.
- Updates to parameters are propagated from the output of the network using the chain rule.
- Update parameters with their gradients and repeat until convergence.
- Use dynamic programming: collect derivatives at each step without recalculating them.
- One epoch is a forward and backward pass over the full-dataset.
- Simple: inputs forward, errors go backward.

Practise: Choices of the Learning Rate

- Choice of the learning rate is critical for the successful training.

$$w \leftarrow w - \eta \frac{d\mathcal{L}}{dw}$$

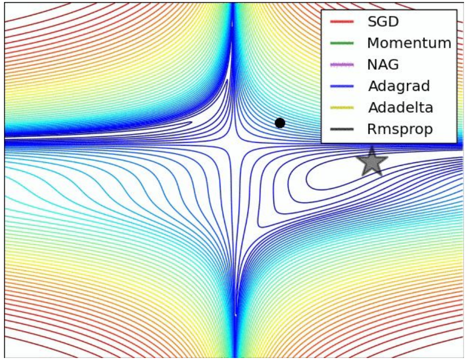


Credit

- There is no fixed rule: depends on the dataset, network.

Gradient Descent Variants

- (mini-batch) Stochastic Gradient Descent (SGD): use random minibatch of examples.
- Adagrad: adjusts the learning rate to individual features.
- Momentum: add a fraction of previous gradient to update vector.
- RMSprop: use a moving average of squared past gradients.
- Adam: RMSProp with momentum and bias correction.



Credit

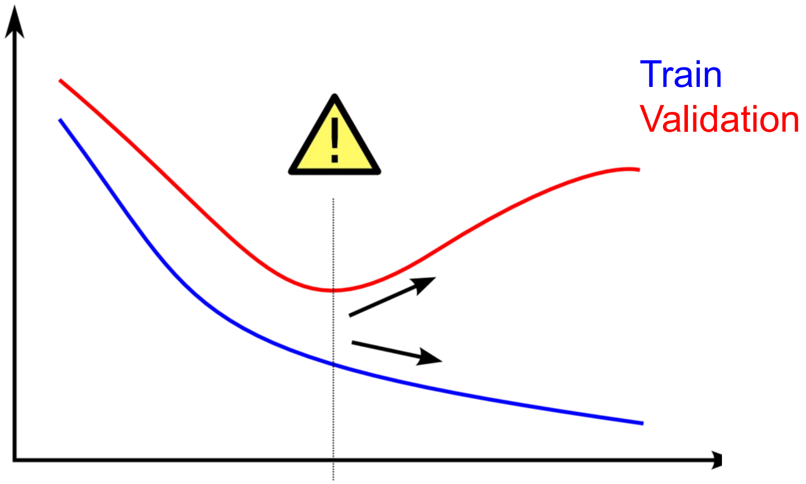
Just use Adam.

Hyperparameters

- Single change in optimization procedure, network architecture or data pre-processing can make or break your model.
- Rules are loose, it is more like art to adjust the hyperparameters.
- How:
  - manual (experience and/or luck),
  - grid searches (random),
  - surrogate models (bayesian optimization, reinforcement learning),
  - specialized software: Ray / AutoKeras.
- What:
  - ~~number of epochs~~,
  - batch size, learning rate,
  - initialization,
  - choice of activation layers, network depth/width (architecture)
  - and many more…

Question Interlude

- Can you describe this situation?



Overfitting

Underfitting (low variance, high bias) with poor train and test results.

- Model is too simple or it can't capture underlying data structure.
- Solution is to increase model capacity or train longer.

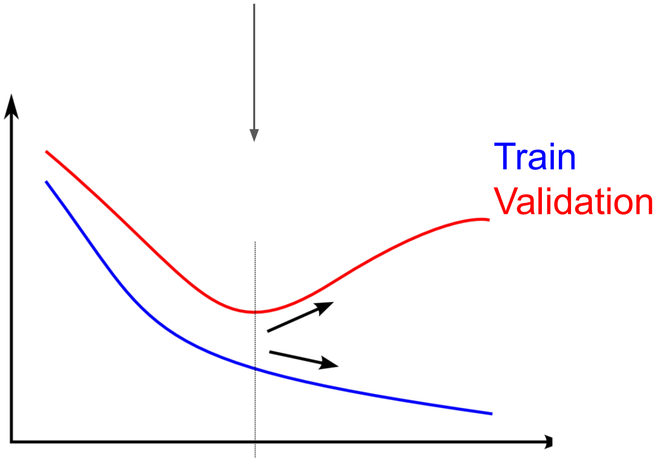
Overfitting (high variance, low bias) with good training error but bad test results.

- Model captures noise instead of the input structure (low generalization).
- Model has too much capacity.
- Solution 1: terminate training before this happens, i.e. early stopping.
- Solution 2: limit capacity of the model by regularization, reduce generalization error but not the training error.
  - Lasso or ridge regularization.
  - Dropout.
  - Data augmentations, transformations of input, e.g. rotations. etc.

Early Stopping

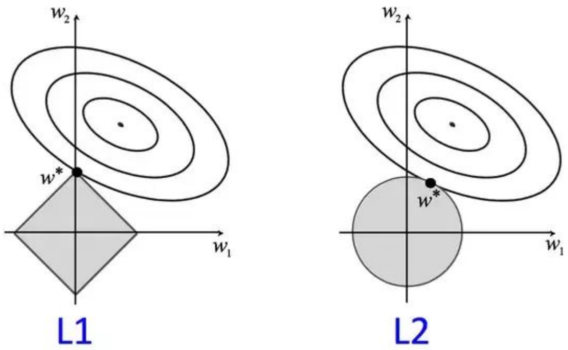
Early stopping is beautiful free lunch Source

- When to stop training? After predefined number of steps? Can be too late or too early.
- Early stopping: stop your training (with some patience) if your validation error does not improve enough.



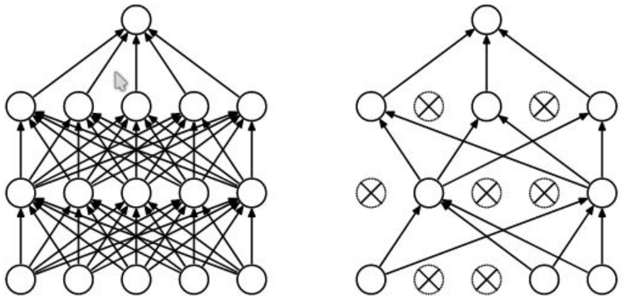
L1/ L2 Regularization

- Limit the capacity of the model by penaling the value of weights.
- L1 regularization (Tikhonov): absolute value of weights for sparsity (feature selection): $\lambda \sum_{i=1}^{n} |\theta_i|$
- L2 regularization (LASSO): penalize square of weights: $\lambda \sum_{i=1}^{n} \theta_i^2$
- λ is a hyperparameter.
- In Pytorch L2 regularization is called `weight decay`.



Dropout (Paper)

- Adding noise to hidden layers makes networks more robust to initialization, and results in better generalization.
- Dropout is a simple way to execute that by randomly set some neuron weights to zero with probability *p*.
- Another interpretation: at each step train a new subnetwork to break co-adaptation of nodes



(a) Standard Neural Net (b) After applying dropout.

Figure 1: Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Batch Normalization (Paper)

- When training, parameters update in different scale, and the initial normalization is lost.
- When the input distribution to a learning system changes, it is said to experience covariate shift.
- Training procedure is sensitive to the scale of gradients:
    - Vanishing gradients: gradients getting smaller and smaller as the backpropagation progresses: no updates.
    - Exploding gradients: gradients getting larger and larger as the backpropagation progresses, very large updates.
- Add an operation just before or after the activation function of each hidden layer.
- The layer lets the model learn the optimal scale and mean of each of the layer's inputs using running mean and standard deviation of the input over the current mini-batch.

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

- Less sensitivity to initialization parameters.

Weight Initialization

- Another way to address vanishing and exploding gradients is through weight initialization.
- We initialize the weights randomly, sampling from a normal distribution, $\mu=0$ and $\sigma=1$.
- This results in a wide range.
- We can ensure that the weights are closer to 0, which works better.
- Xavier initialization normal distribution with $\mu=0$ and $\sigma^2=n^{-1}$, where n is the number of inputs, use it for non-ReLU blocks.
- He initialization: $\mu=0$ and $\sigma^2=2n^{-1}$, use for blocks with ReLU.

Computer Vision

A huge subfield of deep learning dealing with image classification, object detection, segmentation or tracking, depth estimation, 3d reconstruction etc.

Challenges:

- Large dimensionality of the input, e.g. HD image has close to 1M pixels.
- Results must be invariant to shifts, rotation, different light conditions etc.
- Images can contain several objects from multiple categories or multiple instances from one.

Convolutional Neural Networks

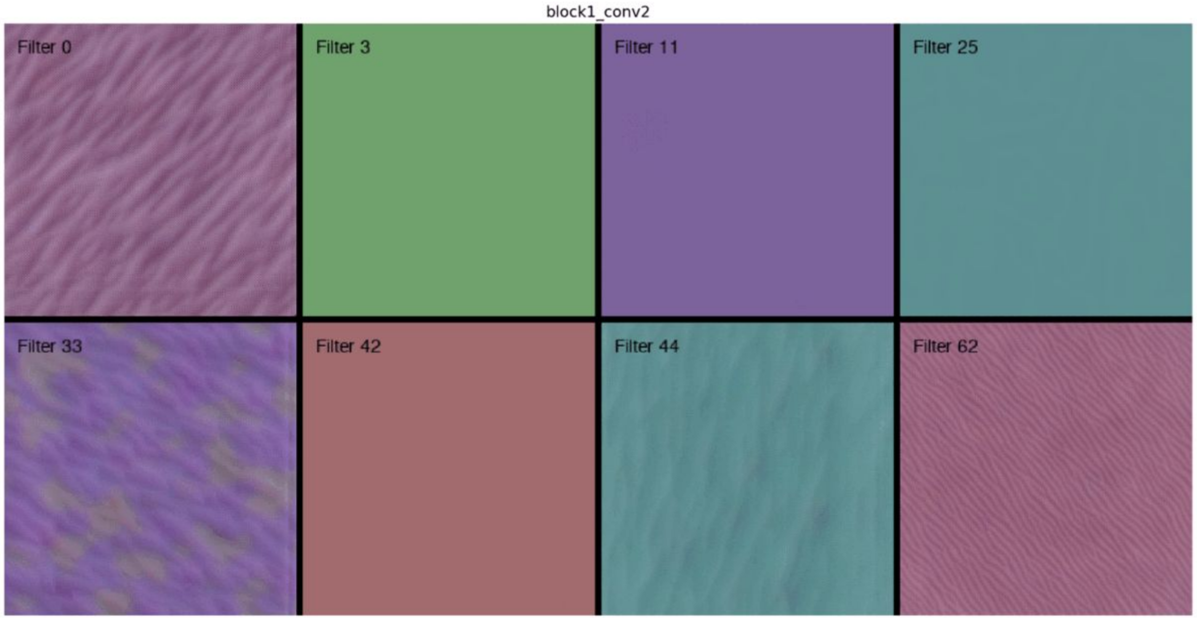
- General idea: sliding window, i.e. slide a matrix (kernel or filter) across input image to check for a specific object (activations will be high).
    - Reduction in trainable parameters through parameter sharing.
    - Location invariance through the use of the sliding window.

- Hand-engineered features are difficult to define.

- We can learn filters instead: create multiple transformed representations of the image and use those as input features to a next layer.

Convolutional Neural Networks

- Each layer becomes more and more expressive.



Convolution Operation

- Output is a dot product between a filter and portion of the input image.
- Hyperparameters: Kernel size, Stride, Padding.

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input

1	0	1
0	1	0
1	0	1

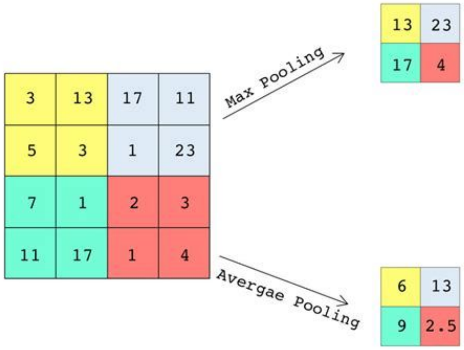
Filter / Kernel

1x1	1x0	1x1	0	0
0x0	1x1	1x0	1	0
0x1	0x0	1x1	1	1
0	0	1	1	0
0	1	1	0	0

4		

Pooling Operation

- Pooling operation down-samples feature maps.
- Two types of operations: averaging (`torch.nn.AvgPool2d`) or max `torch.nn.MaxPool2d`

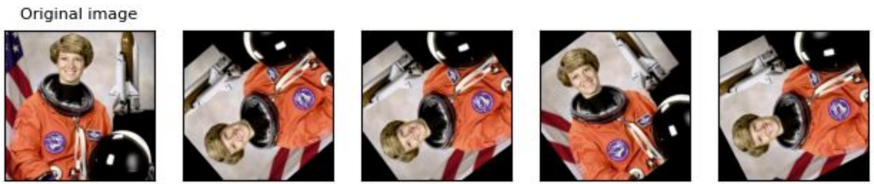


- Lowers computational load of training and inference.
- Avoids overfitting.
- Hyperparameters to chose: type, kernel size and stride.

Data Augmentation

- This is **not** data preprocessing such as image resizing or normalization.
- The performance of CNNs improves with more data.
- If we can't collect more data, we can artificially create new variants of existing **training** data with augmentations.
- Augmentations include a operations such as rotations, shifts, flips, zooms, contrast or hue adjustments.
- You should always consider domain-specific techniques.
- In PyTorch use `torchvision.transforms.Compose`, e.g.:

```
transform=transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465),
                         (0.2023, 0.1994, 0.2010)),
    ])),
```



Transfer Learning

- In practice you won't train a big CNN from random initialization when you have insufficient data points.

- Transfer learning: apply the knowledge that one model holds to a new task.
  - Download a model that has been trained on, for instance, Imagenet (real-world images).
  - Add new layers or adjust existing ones to the shape of your input and output.
  - Train only first and last layer on your data, or more (fine-tuning).

- In PyTorch you can freeze or freeze layers using:

```
param.requires_grad = False
```

- Adjust your learning rate!