Short developer's tutorial on Deep Learning and Neural Networks with PyTorch

Christian Wolf

December 12th, 2019
Decision making

- {dog, cat, bike, avocado, blender, …}
- {0, 1, … 26, 27, 28, …, 98, 99, …}
- {Left, right, ahead, back, pick up, …}
- Motor commands

“A blue parrot with a yellow belly sitting on a branch in a forest”
Deep Learning @ Tesla

https://www.youtube.com/watch?v=oBklltKXtDE

- (3, 960, 1280) input images
- “ResNet-50 like” dilated backbones
- FPN / DeepLabV3 / UNet -like heads
- ~15 tasks => “prototypes framework”
Deep Learning @ Tesla

https://www.youtube.com/watch?v=oBk1ltKXtDE
We would like to **learn** to predict a value $y$ from observed input $x$

$$y = h(x, \theta)$$

- Handcrafted from domain knowledge
- Learned from data or interactions

Fully handcrafted

Fully Learned
The 3 fundamental problems of ML

1. Expressivity
   - What is the complexity of functions my model can represent?

2. Trainability
   - How easy is it to fit my model to my training data?

3. Generalization
   - Does my model generalize to unseen data?
   - In presence of shifts in distributions?

(After Eric Jang & Jascha Sohl-Dickstein)
1. Neural Networks & learning
2. Optimization / gradient descent
3. A simple linear example in PyTorch
4. A multi-layer model in PyTorch
5. Convolutional networks
Deep neural networks

\[ y_k(x) = \sigma \left( \sum_j w_{kj}^{(2)} \left( \sum_i w_{ji}^{(1)} x_i \right) \right) \]

Elementwise activation functions

Input layer \( w^{(1)} \), Hidden layer \( w^{(2)} \), Output layer \( y, y^* \), Loss function \( \mathcal{L} \)
Complex networks

[Baradel, Neverova, Wolf, Mille, Mori, ECCV 2018]
Common activation functions

Hyperbolic tangent
\[ \text{tanh} \]

Rectified Linear Unit
\[ \text{ReLU} \]
Universal approximation

We can approximate any $\psi \in \mathcal{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

Slide: François Fleuret, IDIAP/EPFL
https://www.idiap.ch/~fleuret/
We can approximate any \( \psi \in \mathcal{C}([a, b], \mathbb{R}) \) with a linear combination of translated/scaled ReLU functions.

\[
f(x) = \sigma(w_1 x + b_1)
\]
We can approximate any $\psi \in \mathcal{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2)$$
We can approximate any $\psi \in \mathcal{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1x + b_1) + \sigma(w_2x + b_2) + \sigma(w_3x + b_3)$$
We can approximate any \( \psi \in \mathcal{C}([a, b], \mathbb{R}) \) with a linear combination of translated/scaled ReLU functions.

\[
f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots
\]
We can approximate any $\psi \in C([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots$$
We can approximate any $\psi \in C([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots$$
We can approximate any $\psi \in C([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

\[ f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots \]
We can approximate any \( \psi \in \mathcal{C}([a, b], \mathbb{R}) \) with a linear combination of translated/scaled ReLU functions.

\[
f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots
\]
We can approximate any $\psi \in \mathcal{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots$$
We can approximate any $\psi \in \mathcal{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots$$
We can approximate any $\psi \in \mathcal{C}([a, b], \mathbb{R})$ with a linear combination of translated/scaled ReLU functions.

$$f(x) = \sigma(w_1 x + b_1) + \sigma(w_2 x + b_2) + \sigma(w_3 x + b_3) + \ldots$$

This is true for other activation functions under mild assumptions.
Demo: Tensorflow Playground

https://playground.tensorflow.org
1. Neural Networks & learning

2. Optimization / gradient descent

3. A simple linear example in PyTorch

4. A multi-layer model in PyTorch

5. Convolutional networks
Learning by gradient descent

Iterative minimisation through gradient descent:

\[ \theta^{[t+1]} = \theta^{[t]} + \nu \nabla \mathcal{L}(h(x, \theta), y^*) \]

Can be blocked in a local minimum (not that it matters much …)

[Figure: C. Bishop, 2006]
Gradient descent

One optimizer step:

$$\theta^{[t+1]} = \theta^{[t]} + \nu \nabla \mathcal{L} (h(x, \theta), y^*)$$

The gradient is a vector of partial derivatives:

$$\nabla \mathcal{L} = \begin{bmatrix}
\frac{\partial \mathcal{L}}{\partial \theta_0} \\
\frac{\partial \mathcal{L}}{\partial \theta_1} \\
\vdots \\
\frac{\partial \mathcal{L}}{\partial \theta_N}
\end{bmatrix}$$
Differentiate a multi-layer network

Consider a multi-layer network, in particular an arbitrary unit indexed by $j$ and receiving inputs from units indexed by $i$, providing outputs $z_i$.

Its gradient is:

$$\frac{\partial L}{\partial w_{ji}} = \frac{\partial L}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} = \delta_j z_i$$

$\delta_j$ can be obtained with a recurrence:

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$$

[C. Bishop, Pattern Recognition and Machine Learning, 2006]
The full backpropagation algorithm

1. **Forward pass**: stimulate the model with input $x$, calculate all $a_j$ and $z_j$ up to the output $y$.

2. Calculate the $\delta_j$ for the output units using the derivative of the loss function.

3. **Backward pass**: calculate all $\delta_j$ with

   \[
   \delta_j = h'(a_j) \sum_k w_{kj} \delta_k
   \]

4. Calculate the gradients with

   \[
   \frac{\partial L}{\partial w_{ji}} = \delta_j z_i
   \]
Backpropagation in PyTorch
In PyTorch (and some other frameworks), Autograd performs automatic differentiation through a sequence of tensor instructions of an imperative language.

Let’s consider a simple linear operation:

\[ w = \begin{bmatrix} 5 & 3 \end{bmatrix}, \quad x = \begin{bmatrix} 7 & 2 \end{bmatrix}, \quad y = wx^T \]

The gradient of \( y \) w.r.t to \( x \) is given as

\[ \nabla = \begin{bmatrix} \frac{\partial y}{\partial x_i} \end{bmatrix} = \begin{bmatrix} 5 \\ 3 \end{bmatrix} \]

The gradient of \( y \) w.r.t to \( w \) is given as

\[ \nabla = \begin{bmatrix} \frac{\partial y}{\partial w_i} \end{bmatrix} = \begin{bmatrix} 7 \\ 1 \end{bmatrix} \]
In PyTorch, we will first create the tensors:

```python
w = torch.tensor([5, 3], dtype=float, requires_grad=True)
x = torch.tensor([7, 1], dtype=float, requires_grad=True)
```

The `requires_grad` flag ensures that all calculations are tracked. We perform the linear operation:

```python
y = torch.dot(w, x)
```

Since the tensor `y` has been calculated as result of operations on tracked tensors, it has a gradient function:

```python
print(y)
tensor(38., dtype=torch.float64, grad_fn=<DotBackward>)
```
We now run a backward pass on the variable $y$, which calculates gradients w.r.t. to all involved tensors:

```
y.backward()
```

The gradients are attached to each variable:

```
print (x.grad)
print (w.grad)
```

```
tensor([5., 3.], dtype=torch.float64)
tensor([7., 1.], dtype=torch.float64)
```
1. Neural Networks & learning

2. Optimization / gradient descent

3. A simple linear example in PyTorch

4. A multi-layer model in PyTorch

5. Convolutional networks
Example: Wisconsin breast-cancer

<table>
<thead>
<tr>
<th>Id</th>
<th>Cl.thickness</th>
<th>Cell.size</th>
<th>Cell.shape</th>
<th>Marg.adhesion</th>
<th>Epith.c.size</th>
<th>Bare.nuclei</th>
<th>Bl.cromatin</th>
<th>Normal.nucleoli</th>
<th>Mitoses</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000025</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>benign</td>
</tr>
<tr>
<td>1002945</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>10</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>benign</td>
</tr>
<tr>
<td>1015425</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>benign</td>
</tr>
<tr>
<td>1016277</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>benign</td>
</tr>
<tr>
<td>1017023</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>benign</td>
</tr>
<tr>
<td>1017122</td>
<td>8</td>
<td>10</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>10</td>
<td>9</td>
<td>7</td>
<td>1</td>
<td>malignant</td>
</tr>
<tr>
<td>1018099</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>benign</td>
</tr>
<tr>
<td>1018561</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>benign</td>
</tr>
<tr>
<td>1033078</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>benign</td>
</tr>
<tr>
<td>1033078</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>benign</td>
</tr>
<tr>
<td>1035283</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>benign</td>
</tr>
<tr>
<td>1036172</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>benign</td>
</tr>
<tr>
<td>1041801</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>malignant</td>
</tr>
</tbody>
</table>

https://www.machinelearningplus.com/machine-learning/logistic-regression-tutorial-examples-r/
import numpy as np
from numpy import genfromtxt
import torch
from torch.nn import functional as F

# Import the text file into a numpy array
n = genfromtxt('breast-cancer-wisconsin-cleaned.csv',
               delimiter=',',)
D = torch.tensor(n, dtype=torch.float32)
N_samples = D.size(0)

# The input is the full matrix without first and
# last column, plus the 1 column for the bias
X=D[:,1:-1]
X = torch.cat ((X, torch.ones((X.size(0),1))),1)

# The targets. Change all 2->0 and 4->1
T=D[:,1:-1]
T[T==2]=0
T[T==4]=1
class LogisticRegression(torch.nn.Module):
    def __init__(self):
        super(LogisticRegression, self).__init__()

        # The linear layer (input dim, output dim)
        # It also contains a weight matrix
        # (here single output -> vector)
        self.fc1 = torch.nn.Linear(10, 1)

        # The forward pass of the network. x is the input
    def forward(self, x):
        return F.sigmoid(self.fc1(x))
# Instantiate the model
model = LogisticRegression()

# The loss function: binary cross-entropy
criterion = torch.nn.BCELoss()

# Set up the optimizer: stochastic gradient descent
# with a learning rate of 0.01
optimizer = torch.optim.SGD(model.parameters(), lr =0.01)
Iterative training

# 1 epoch = 1 pass over the full dataset
for epoch in range(200):
    print ("Starting epoch", epoch, "", end='')
calcAccuracy()

    for sample in range(N_samples):
        # model -> train mode, clear gradients
        model.train()
        optimizer.zero_grad()

        # Forward pass (stimulate model with inputs)
        y = model(X[sample,:])

        # Compute Loss
        loss = criterion(y, T[sample])

        # Backward pass: calculate the gradients
        loss.backward()

        # One step of stochastic gradient descent
        optimizer.step()
Calculate the accuracy (in percent) at each epoch:
Proportion of correctly classified samples.
Random performance = 50% on a binary task.

```python
def calcAccuracy():
    # model -> eval mode
    model.eval()
    correct = 0.0
    for sample in range(N_samples):

        # threshold the output probability
        y = 1 if model(X[sample, :]) > 0.5 else 0
        correct += (y == T[sample]).numpy()

    print("Accuracy = ", 100.0*correct/N_samples)
```
<table>
<thead>
<tr>
<th>Starting epoch</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[34.69985359]</td>
</tr>
<tr>
<td>1</td>
<td>[91.21522694]</td>
</tr>
<tr>
<td>2</td>
<td>[93.99707174]</td>
</tr>
<tr>
<td>3</td>
<td>[95.16837482]</td>
</tr>
<tr>
<td>4</td>
<td>[95.75402635]</td>
</tr>
<tr>
<td>5</td>
<td>[95.60761347]</td>
</tr>
<tr>
<td>6</td>
<td>[95.75402635]</td>
</tr>
<tr>
<td>7</td>
<td>[95.75402635]</td>
</tr>
<tr>
<td>8</td>
<td>[96.33967789]</td>
</tr>
<tr>
<td>9</td>
<td>[96.48609078]</td>
</tr>
<tr>
<td>10</td>
<td>[96.63250366]</td>
</tr>
<tr>
<td>11</td>
<td>[96.63250366]</td>
</tr>
<tr>
<td>12</td>
<td>[96.63250366]</td>
</tr>
<tr>
<td>13</td>
<td>[96.63250366]</td>
</tr>
<tr>
<td>14</td>
<td>[96.63250366]</td>
</tr>
<tr>
<td>15</td>
<td>[96.63250366]</td>
</tr>
</tbody>
</table>

(...)

<table>
<thead>
<tr>
<th>Starting epoch</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>197</td>
<td>[97.07174231]</td>
</tr>
<tr>
<td>198</td>
<td>[97.07174231]</td>
</tr>
<tr>
<td>199</td>
<td>[97.07174231]</td>
</tr>
</tbody>
</table>
What is missing?

— The model is simpler than deep neural networks, but sufficient for the task.
— We did not use batch processing, i.e. using more than one sample for a given gradient update
— We calculated performance on the training set. We might overfit.
1. Neural Networks & learning
2. Optimization / gradient descent
3. A simple linear example in PyTorch
4. A multi-layer model in PyTorch
5. Convolutional networks
Example: the MNIST dataset

A dataset of handwritten digits introduced by Yann LeCun in 1999 with 60,000 training images and 10,000 test images. One image is of size 28x28 pixels.

http://yann.lecun.com/exdb/mnist/
Writing Data Access

We subclass the `Dataset` class and implement the methods `__getitem__`, `__len__()` (and `__init__`)!). This class will be used by actual PyTorch DataLoader, `DataLoader`.

```python
import torch
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms

class MNISTDataset(Dataset):
    def __init__(self, dir, transform=None):
        # Read in the files from the directory and
        # store them in self.images and self.labels
        # See MOODLE for the full code of this method (...)

        # The access is _NOT_ shuffled. The PyTorch
        # Dataloader will need to do this.
        def __getitem__(self, index):
            return self.images[index], self.labels[index]

        # Return the dataset size
        def __len__(self):
            return self.no_images
```
Writing Data Access

Let’s recall training through gradient descent:

\[ \theta^{[t+1]} = \theta^{[t]} + \nu \nabla \mathcal{L}(h(x, \theta), y^*) \]

The gradient is rarely (never?) taken over the whole dataset, but over a single sample, or batches (mini-batches) of a certain size. These batches are sampled randomly from the dataset.

The actual shuffling and batching is performed by a built-in PyTorch DataLoader class, which uses an instance of our Dataset subclass:

```python
dataset = MNISTDataset("MNIST-png/testing",
    transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ]))
loader = torch.utils.data.DataLoader(dataset,
    batch_size=50, shuffle=True)
```

We passed a set of image transforms to the Dataset class, which applies them to each image.
Tensor dimension conventions

PyTorch functions operate on multi-dimensional tensors and follow conventions on the order of dimensions.

— The first dimension is the batch dimension
  ▶ Use 1 if you don’t use batches (= batches of size 1).
  ▶ Losses are reduced (sum or mean) over samples in a batch

— the second dimension is the channel dimension
  ▶ Use 1 if you don’t use channels (= single channels).
  ▶ Channel arithmetics will be explained in detail in the section on convolutions.

— the following dimensions are application dependant, e.g. rows, columns in images.
The 2 layer model

The model is similar to our linear example. Differences:

— A hidden layer with 300 units and relu activation.
— A forward pass deals with a full batch.

```python
class MLP(torch.nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        # input size to 300 units
        self.fc1 = torch.nn.Linear(28*28, 300)
        # 300 units to 10 output classes
        self.fc2 = torch.nn.Linear(300, 10)

    def forward(self, x):
        # Reshape from a 3D tensor (batchsize, 28, 28)
        # to a flattened (batchsize, 28*28)
        # 1 sample = 1 vector
        x = x.view(-1, 28*28)
        x = F.relu(self.fc1(x))
        return self.fc2(x)
```
This time we have more than 2 classes, we use the Cross-entropy loss.

```python
# Instantiate the model
model = MLP()

# This criterion combines LogSoftMax and NLLLoss
# in one single class.
crossentropy = torch.nn.CrossEntropyLoss()

# Set up the optimizer: stochastic gradient descent
# with a learning rate of 0.01
optimizer = torch.optim.SGD(model.parameters(), lr = 0.01)

# Init some statistics
running_loss = 0.0
running_correct = 0
running_count = 0
```
Cycling through batches and samples

```python
# Cycle through epochs
for epoch in range(100):
    # Cycle through batches. One batch is a set of
    # images and a set of ground truth labels
    for batch_idx, (data, labels) in enumerate(loader):
        optimizer.zero_grad()

        # We calculate a prediction on the full batch
        y = model(data)

        # Loss of the full batch, summed
        loss = crossentropy(y, labels)

        # Calculate gradients of the full batch
        loss.backward()

        # One gradient update
        optimizer.step()
```
# Calculate the winner class
_, predicted = torch.max(y.data, 1)

# How many correct samples?
running_correct += (predicted == labels).sum().item()
running_count += BATCHSIZE

# Every 100 batches, print statistics
if (batch_idx % 100) == 0:
    train_err = 100.0*(1.0 - running_correct / running_count)
    print ('Epoch: %d batch: %5d ' % (epoch + 1, batch_idx + 1), end="""
    print ('train-loss: %.3f train-err: %.3f' % (running_loss / 100, train_err))
    running_loss = 0.0
    running_correct = 0.0
    running_count=0.0
### Example output

<table>
<thead>
<tr>
<th>Epoch</th>
<th>Batch</th>
<th>Loss</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0.024</td>
<td>98.000</td>
</tr>
<tr>
<td>2</td>
<td>101</td>
<td>1.576</td>
<td>36.140</td>
</tr>
<tr>
<td>3</td>
<td>201</td>
<td>0.747</td>
<td>16.320</td>
</tr>
<tr>
<td>4</td>
<td>301</td>
<td>0.539</td>
<td>12.880</td>
</tr>
<tr>
<td>5</td>
<td>401</td>
<td>0.481</td>
<td>13.000</td>
</tr>
<tr>
<td>6</td>
<td>501</td>
<td>0.414</td>
<td>11.280</td>
</tr>
<tr>
<td>7</td>
<td>601</td>
<td>0.386</td>
<td>10.380</td>
</tr>
<tr>
<td>8</td>
<td>701</td>
<td>0.385</td>
<td>10.900</td>
</tr>
<tr>
<td>9</td>
<td>801</td>
<td>0.363</td>
<td>10.540</td>
</tr>
<tr>
<td>10</td>
<td>901</td>
<td>0.320</td>
<td>9.120</td>
</tr>
<tr>
<td>11</td>
<td>1001</td>
<td>0.323</td>
<td>8.920</td>
</tr>
<tr>
<td>12</td>
<td>1101</td>
<td>0.325</td>
<td>9.400</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>0.304</td>
<td>8.880</td>
</tr>
</tbody>
</table>

(...)

<table>
<thead>
<tr>
<th>Epoch</th>
<th>Batch</th>
<th>Loss</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>75</td>
<td>801</td>
<td>0.007</td>
<td>0.000</td>
</tr>
<tr>
<td>75</td>
<td>901</td>
<td>0.007</td>
<td>0.020</td>
</tr>
<tr>
<td>75</td>
<td>1001</td>
<td>0.008</td>
<td>0.000</td>
</tr>
<tr>
<td>75</td>
<td>1101</td>
<td>0.009</td>
<td>0.040</td>
</tr>
</tbody>
</table>

This is training error, not validation error, i.e. **NOT** representative of the performance of the model!
1. Neural Networks & learning
2. Optimization / gradient descent
3. A simple linear example in PyTorch
4. A multi-layer model in PyTorch
5. Convolutional networks
Fully connected layers might be harmful

400x400 pixels

Flattened into 160,000 values

500 units

500 * 160,000 = 80 Million parameters (for a single layer!)
Fully connected layers might be harmful

Parameters learned for one part of the image do not generalize to other parts of the image.
Shift-invariant linear operations

Classical MLPs are sequences of linear layers followed by point-wise non-linearities. What kind of linear and shift-invariant operators can there exist?

Let's consider an operator $\mathcal{O}$ with following properties:

**Linearity:**

$$\phi (\alpha f + \beta f') = \alpha \phi(f) + \beta \phi(f')$$

**Shift-invariance:**

$$\phi (^{m,n}S(f)) = ^{m,n}S(\phi(f))$$

where $^{m,n}S(f)$ shifts a signal by $^{m,n}$.

**Impulse response** $h$:

$$h = \phi(0,0p)$$

where $0,0p$ is a Dirac impulse centered at $0,0$. 
Convolutions

(Each value is divided by 16)
Convolutions

(Each value is divided by 16)

\[
\frac{(3 + 2 \times 5 + 3 + 2 \times 4 + 5 \times 5 + 2 \times 7 + 3 + 2 \times 7 + 1)}{16} = 5
\]
Convolutions

Each value is divided by 16

\[
\frac{3+2*5+3+2*4+5*5+2*7 +3+2*7+1}{16} = 5
\]
Convolutions

Each value is divided by 16.

(Each value divided by 16)

\( \phi(f) \)
Convolutions

\[
\phi(f) = \frac{(5 + 2 \times 3 + 17 + 2 \times 5 + 4 \times 7 + 2 \times 18 + 7 + 2 \times 1 + 16)}{16} = 7
\]
"LeNet", by Yann LeCun, is a network composed of sequences of convolutions and spatial size reductions (« pooling »).

[LeCun et al., 1998]
Each of the M output channels is a linear combination of the N filtered input images.

We need $N \times M$ filters!
Template matching

Input image:

• We normalize input and kernel (subtract mean, divide by standard deviation).
• We convolve the input with the kernel
• We threshold response and superimpose it on the red channel.
Convolutions can match patterns:
• either in input space
• or in intermediate network layers (which still have a spatial organization).
class LeNet(torch.nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        # 1 input channel, 20 output channels,
        # 5x5 filters, stride=1, no padding
        self.conv1 = torch.nn.Conv2d(1, 20, 5, 1, 0)
        self.conv2 = torch.nn.Conv2d(20, 50, 5, 1, 0)
        self.fc1 = torch.nn.Linear(4*4*50, 500)
        self.fc2 = torch.nn.Linear(500, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        # Max pooling with a filter size of 2x2
        # and a stride of 2
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 4*4*50)
        x = F.relu(self.fc1(x))
        return self.fc2(x)
MLP vs LeNet on MNIST

Blue: training, Red: validation
Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

The second convolutional layer takes as input the (response-normalized and pooled) output of the first convolutional layer and filters it with 256 kernels of size $5 \times 48$.

The third, fourth, and fifth convolutional layers are connected to one another without any intervening pooling or normalization layers. The third convolutional layer has 384 kernels of size $3 \times 3 \times 256$ connected to the (normalized, pooled) outputs of the second convolutional layer. The fourth convolutional layer has 384 kernels of size $3 \times 3 \times 192$, and the fifth convolutional layer has 256 kernels of size $3 \times 3 \times 192$. The fully-connected layers have 4096 neurons each.

4 Reducing Overfitting

Our neural network architecture has 60 million parameters. Although the 1000 classes of ILSVRC make each training example impose 10 bits of constraint on the mapping from image to label, this turns out to be insufficient to learn so many parameters without considerable overfitting. Below, we describe the two primary ways in which we combat overfitting.

4.1 Data Augmentation

The easiest and most common method to reduce overfitting on image data is to artificially enlarge the dataset using label-preserving transformations (e.g., [25, 4, 5]). We employ two distinct forms of data augmentation, both of which allow transformed images to be produced from the original images with very little computation, so the transformed images do not need to be stored on disk. In our implementation, the transformed images are generated in Python code on the CPU while the GPU is training on the previous batch of images. So these data augmentation schemes are, in effect, computationally free.

The first form of data augmentation consists of generating image translations and horizontal reflections. We do this by extracting random $224 \times 224$ patches (and their horizontal reflections) from the $256 \times 256$ images and training our network on these extracted patches. This increases the size of our training set by a factor of 2048, though the resulting training examples are, of course, highly interdependent. Without this scheme, our network suffers from substantial overfitting, which would have forced us to use much smaller networks. At test time, the network makes a prediction by extracting five $224 \times 224$ patches (the four corner patches and the center patch) as well as their horizontal reflections (hence ten patches in all), and averaging the predictions made by the network's softmax layer on the ten patches.

The second form of data augmentation consists of altering the intensities of the RGB channels in training images. Specifically, we perform PCA on the set of RGB pixel values throughout the ImageNet training set. To each training image, we add multiples of the found principal components.

4 Going deeper

2012: AlexNet, 8 layers. New techniques: dropout, ReLU


2015: Microsoft research, 150 layers. New technique: residual learning
Visualizing feature map activations

Layer 1

Layer 2

[Zeiler and Fergus, ECCV 2014]
Visualizing feature map activations

Layer 3

[Zeiler and Fergus, ECCV 2014]
Visualizing feature map activations

Layer 2
Layer 3
Layer 4
Layer 5

[Zeiler and Fergus, ECCV 2014]
Conclusion

Problems to solve:
- Expressivity: model symmetries and invariances in your data
- Trainability
- Generalization

1960-2012: let's handcraft features
2012-2016: let's handcraft architectures (layers, units ...)
2016- ?: let's handcraft data-flow (geometry, causality, attention)