Introduction to Deep Learning

Computer Vision CMP-6035B

Dr. David Greenwood

david.greenwood@uea.ac.uk

SCI 2.16a University of East Anglia

Spring 2022

Content

- ImageNet
- Neural Networks
- MNIST Examples
- Convolutional Neural Networks

ImageNet

- $>$ 1,000,000 images
- $>$ 1,000 classes

Actually...

$>$ 15,000,000 images

$>$ 20,000 classes

Ground truth annotated manually with Amazon *Mechanical Turk*.

Freely available for research here: https://www.image-net.org/

Figure 1: mushrooms

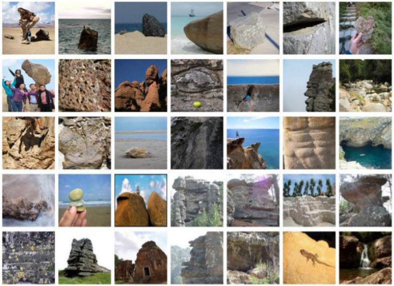

Figure 2: landscape

ImageNet Top-5 challenge:

You score if ground truth class is one your top 5 predictions!

ImageNet in 2012

- Best approaches used hand-crafted features.
- SIFT, HOGs, Fisher vectors, etc. plus a classifier.
- Top-5 error rate: $\sim$25%

Then the game changed!

AlexNet

In 2012, Krizhevsky et al. used a deep neural network to achieve a **15%** error rate.

- AlexNet
- Five convolutional layers...
- ...followed by three fully connected layers.
- ImageNet Classification with Deep Convolutional Neural Networks.

Prior approaches used hand *designed* features.

Neural networks **learn** features that help them classify and quantify images.

Neural Networks

What *is* a neural network?

Neural Networks

Multiple *layers*.

Data *propagates* through layers.

Transformed by each layer.

Neural Network Classifier

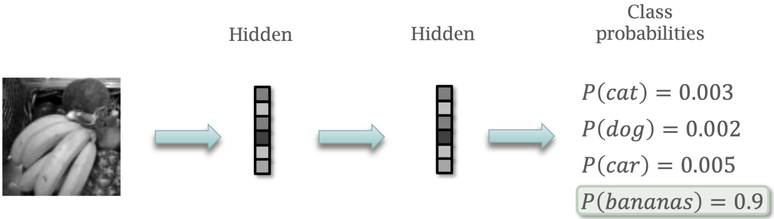

Figure 3: Neural Network for classification

Neural Network Regressor

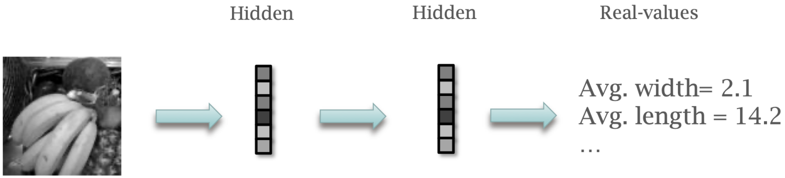

Figure 4: Neural Network for regression

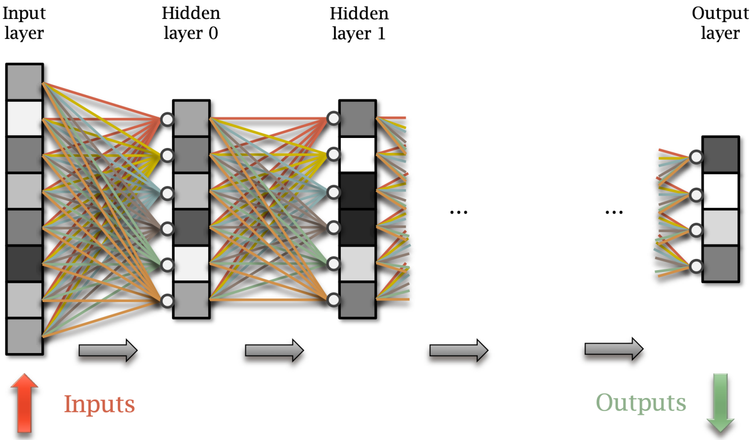

Figure 5: Neural Network Weights

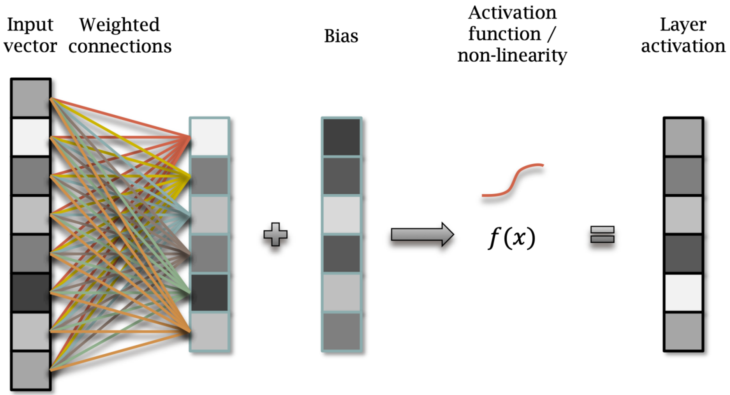

Figure 6: Single Layer

- x input vector of size M
- y output vector of size N
- W weight matrix of size $M \times N$
- b bias vector of size N
- f activation function, e.g. ReLU: $\max(x, 0)$

$$y = f(Wx + b)$$

$$y = f(Wx + b)$$

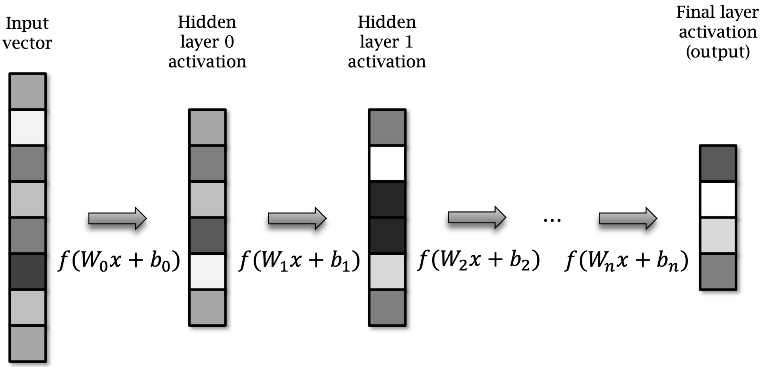

Figure 7: Multiple Layers

$$
\begin{aligned}
y_0 &= f(W_0 x + b_0) \\
y_1 &= f(W_1 y_0 + b_1) \\
&\cdots \\
y_L &= f(W_L y_{L-1} + b_L)
\end{aligned}
$$

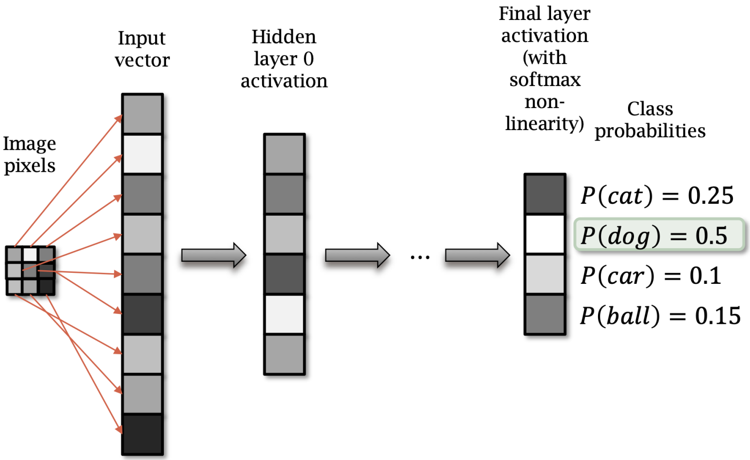

Figure 8: Classifier Layers

A **Neural Network** is built from *layers*, each of which is:

- a matrix multiplication
- a bias
- a non-linear activation function

Practical Examples

. . . using **PyTorch**.

Practical Examples

I've provided a small repository of code examples for you to try out, at: https://github.com/uea-teaching/Deep-Learning-for-Computer-Vision

Figure 9: Code Examples

Practical Examples

The first thing to note, is we usually work with **batches** of input data.

- or, more strictly, *mini-batches*.
- For a sample of M values, then a mini-batch of S samples is an S x M matrix.

```python
import torch, torch.nn.functional as F

# Assume input_data is S * M matrix
x = torch.tensor(input_data)

# W: gaussian random M * N matrix, std-dev=1/sqrt(N)
W = torch.randn(M, N) / math.sqrt(N)

# Bias: zeros, N elements
b = torch.zeros(1, N)

y = F.relu(x @ W + b)
```

This is all a bit clunky.

PyTorch provides nice convenient layers for you to use.

```python
# Assume input_data is S * M matrix
x = torch.tensor(input_data)

# Linear layer, M columns in, N columns out
layer = torch.nn.Linear(M, N)

# Call the layer like a function to apply it
y = F.relu(layer(x))
```

Training

On order to *learn* the correct weights, we need to **train** the model.

Training

Define a **cost** to measure the *error* between predictions and ground truth.

Training

Use **back-propagation** to modify *parameters* so that cost drops toward zero.

Initialisation

Initialise weights randomly.

- We can follow the scheme proposed by He, et al. in 2015.
- We did this earlier, the scaled random normal initialisation.
- Pytorch does this by default, so no need to worry about it.

Training

For each example x_{train} from the training set.

- *Evaluate* y_{pred} given the training input.
- Measure the *cost*: $c = (y_{pred} - y_{train})$
- Iteratively reduce the cost using **gradient descent**.

Compute the derivative of *cost* c

- w.r.t. all parameters W and b.

Update parameters W and b using gradient descent:

$$W_0' = W_0 - \lambda \frac{\partial c}{\partial W_0}$$
$$b_0' = b_0 - \lambda \frac{\partial c}{\partial b_0}$$

λ is the learning rate: a *hyperparameter*.

Theoretically. . . use the chain rule to calculate gradients.

– This is time consuming.
– Easy to make mistakes.

In Practice

Many Neural Network tool-kits do all this for you automatically.

Write the code that performs the **forward** operations, PyTorch keeps track of what you did and will compute *all* the gradients in one step!

Computing gradients in PyTorch

```
# Get predictions, no non-linearity
y_pred = layer(x_train)
# Cost is mean squared error
cost = ((y_pred - y_train) ** 2).mean()
# Compute gradients using 'backward' method
cost.backward()
```

Gradient descent in PyTorch

```
# Create an optimizer to update the parameters of layer
opt = torch.optim.Adam(layer.parameters(), lr=1e-3)

# Get predictions and cost as before
y_pred = layer(x_train)
cost = ((y_pred - y_train) ** 2).mean()
# Back-prop, zero the gradients attached to params first
opt.zero_grads()
# compute gradients
cost.backward()
# update the parameters
opt.step()
```

Classification

Final layer has a **softmax** non-linear function.

The cost is the cross-entropy loss, which is the negative log-likelihood.

Softmax

Softmax produces a probability vector:

$$q(x) = \frac{e^{x_i}}{\sum_{i=0}^{N} e^{x_i}}$$

Classification Cost

Negative log probability (categorical cross-entropy):

- q is the predicted probability.
- p is the true probability (usually 0 or 1).

$$c = -\sum p_i \log q_i$$

Classification in PyTorch

```
# Create a nn.CrossEntropyLoss object to compute loss
criterion = torch.nn.CrossEntropyLoss()
# Get predicted logits
y_pred_logits = layer(x_train)
# Use criterion to compute loss
cost = criterion(y_pred_logits, y_train)
...
```

Regression

To quantify something, with real-valued output.

Cost: Mean squared error.

Mean Squared Error

- q is the predicted value.
- p is the true value.

$$c = \frac{1}{N} \sum_{i=0}^{N} (q_i - p_i)^2$$

Regression in PyTorch

```
# Create a nn.CrossEntropyLoss object to compute loss
criterion = torch.nn.MSELoss()
# Get predicted logits
y_pred_logits = layer(x_train)
# Use criterion to compute loss
cost = criterion(y_pred_logits, y_train)
...
```

Training

Randomly split the training set into mini-batches of approximately 100 samples.

- Train on a mini-batch in a single step.
- The mini-batch cost is the mean of the costs of all samples in the mini-batch.

Training on mini-batches means that $\sim$100 samples are processed in parallel.

- Good news for GPUs that do lots of operations in parallel.

Training on enough mini-batches to cover all examples in the training set is called an epoch.

- Run multiple epochs (often 200-300), until the cost converges.

Training - Recap

1. Take mini-batch of training examples.
2. Compute the cost of the mini-batch.
3. Use gradient descent to update parameters and reduce cost.
4. Repeat, until done.

Multi-Layer Perceptron

The simplest network architecture...

Multi-Layer Perceptron (MLP)

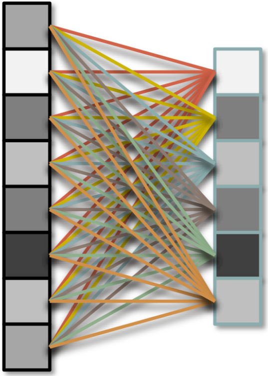

Figure 10: dense layer

Dense layer

Each unit is connected to all units in previous layer.

MNIST Example

The “Hello World” of neural networks.

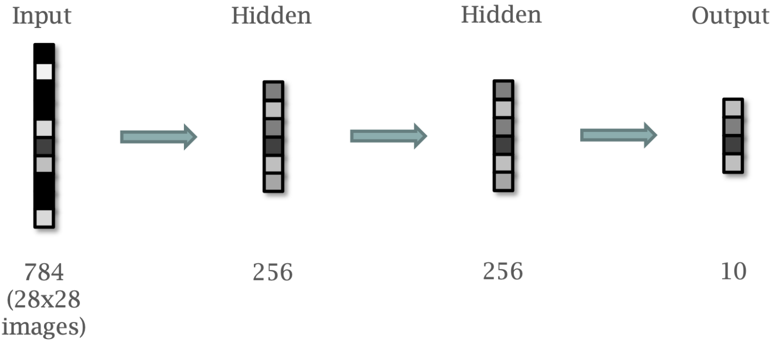

Figure 11: MNIST-MLP

```python
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.input = nn.Linear(784, 256)
        self.hidden = nn.Linear(256, 256)
        self.output = nn.Linear(256, 10)

    def forward(self, x):
        x = x.view(x.shape[0], -1)
        x = F.relu(self.input(x))
        x = F.relu(self.hidden(x))
        return self.output(x)
```

```python
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.input = nn.Linear(784, 256)
        self.hidden = nn.Linear(256, 256)
        self.output = nn.Linear(256, 10)

    def forward(self, x):
        x = x.view(x.shape[0], -1)
        x = F.relu(self.input(x))
        x = F.relu(self.hidden(x))
        return self.output(x)
```

```python
class Model(nn.Module):
    def __init__(self):
        super().__init__()
        self.input = nn.Linear(784, 256)
        self.hidden = nn.Linear(256, 256)
        self.output = nn.Linear(256, 10)

    def forward(self, x):
        x = x.view(x.shape[0], -1)
        x = F.relu(self.input(x))
        x = F.relu(self.hidden(x))
        return self.output(x)
```

MNIST

MNIST is quite a special case.

- Digits nicely centred within the image.
- Scaled to approximately the same size.

Visualisation

Figure 12: MNIST Samples

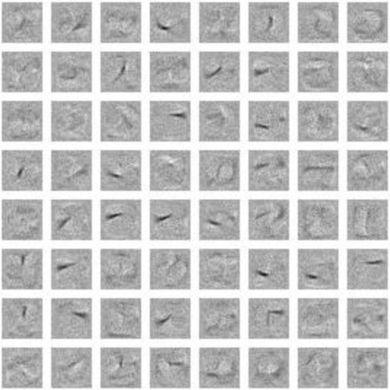

Figure 13: Weight Visualisation

Visualisation

Note the stroke features detected by the various units.

Figure 14: MNIST Samples

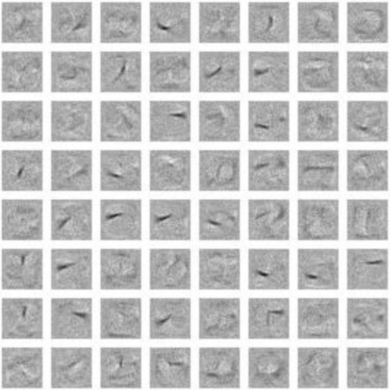

Figure 15: Weight Visualisation

Visualisation

Learned features lack translation invariance.

Figure 16: MNIST Samples

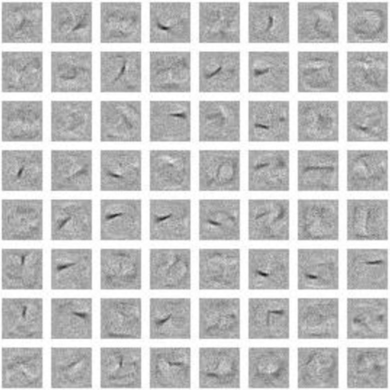

Figure 17: Weight Visualisation

For more general imagery:

- Require a training set large enough to see all features in all possible positions.
- Require network with enough units to represent this.

Convolutional Neural Networks

The computer vision revolution...

Convolution

We have already discussed convolution.

- Slide a filter, or kernel, over the image.
- Multiply image pixels by filter weights and sum.
- Do this for all possible positions of the filter.

Convolution

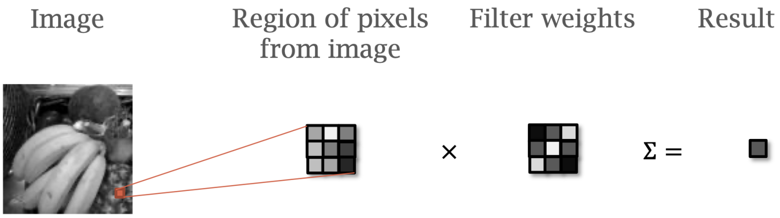

Figure 18: Convolution

Convolution

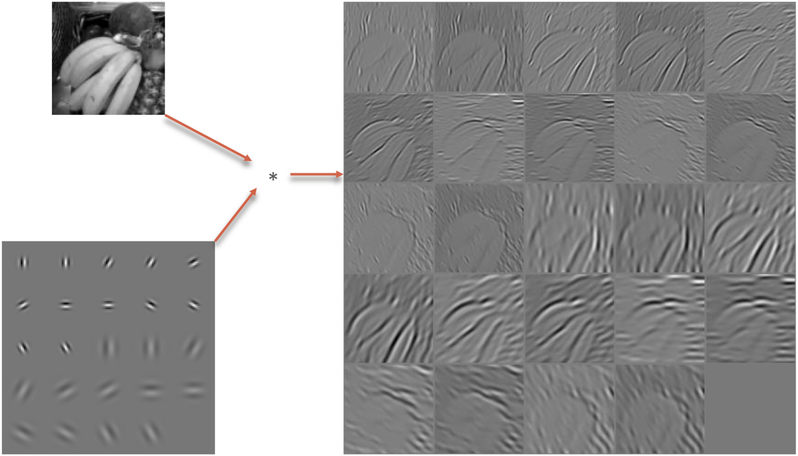

Figure 19: Gabor Filter

Convolution

Convolution detects features in a *position independent* manner.

Convolutional neural networks **learn** position independent filters.

Recap: Fully Connected Layer

Each hidden unit is fully connected to all inputs.

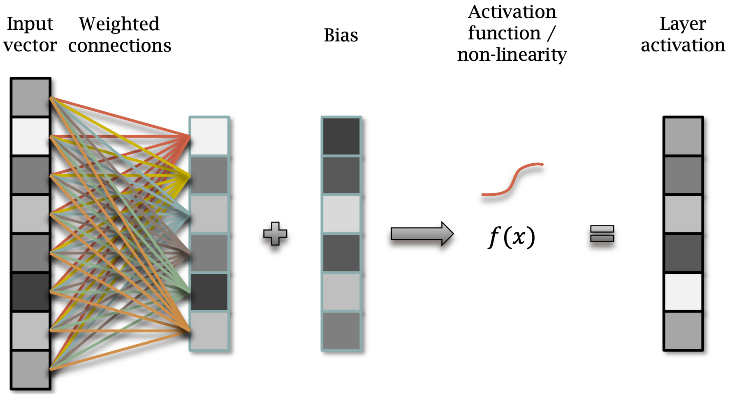

Figure 20: Fully Connected Layer

Convolution

Each hidden unit is only connected to inputs in its local neighbourhood.

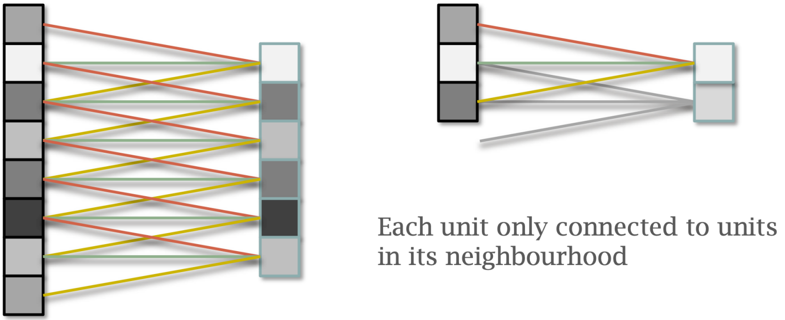

Figure 21: Convolution Detections

Convolution

Each group of weights is shared between all units in the layer.

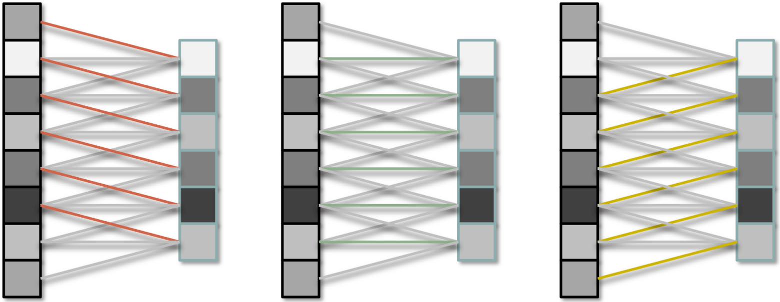

Figure 22: Shared Weights

Convolution

The values of the weights form a **filter**.

For practical computer vision, more than one filter must be used to extract a variety of features.

Convolution

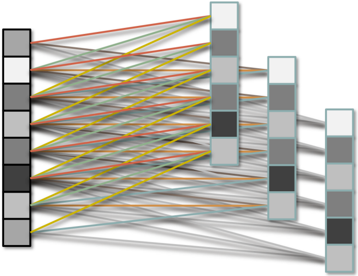

Figure 23: Multiple Filters

Multiple filter weights.
Output is image with multiple channels.

Convolution

Convolution can be expressed as multiplication by weight matrix.

$$y = f(Wx + b)$$

Convolution

In subsequent layers, each filter connects to pixels in **all** channels in previous layer.

Max Pooling

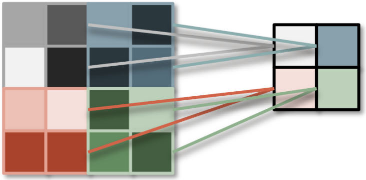

Figure 24: Max Pooling

Take the maximum from each $(p \times p)$ pooling region.
Down sample the image by a factor of p.

Striding

We can also down-sample using **strided** convolution.

- Generate output for 1 in every n pixels.
- Faster, can work as well as max-pooling.

ConvNetJS

Visualisations are avalable at ConvNetJS by Andrej Karpathy.

https://cs.stanford.edu/people/karpathy/convnetjs/index.html

Source code for the site is available at:

https://github.com/karpathy/convnetjs

Summary

- ImageNet
- Neural Networks
- MNIST Examples
- Convolutional Neural Networks