Chapter – 5

**Neural Networks for Machine Learning**

**Geoffrey Hinton**

with **Nitish Srivastava** & **Kevin Swersky**

**Object Recognition &**

**Convolutional Nets**

Lectures: Geoffrey Hinton

Why object recognition is difficult

Achieving viewpoint invariance

Convolutional nets for digit recognition

Convolutional nets for object recognition

**5.1 Why object recognition is difficult**

In this section, we'll discuss some of the things that make it *difficult to recognize objects* in *real scenes*.

* We humans are incredibly good at this, and so it's very hard for us to realize how difficult it is to take a *bunch of* ***numbers*** that *describe* *the* ***intensities of pixels*** and go from those numbers to the *label of an object*.
* There's all sorts of difficulties like:
* We have to segment the object out.
* We have to deal with variations in lighting, in viewpoint.
* We have to deal with the fact that the definitions of objects are quite complicated.
* Recognizing an object from an image *requires huge amounts of* ***knowledge***, even for the *lower level processes* that involve segmentation and dealing with viewpoint and lighting.

If that's the case, it's gonna be very hard for any ***hand engineered program*** to be able to do a good job of those things.

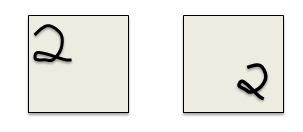
* **Things that make it hard to recognize objects**

There are many reasons why it's hard to recognize objects in images.

* Segmentation: First of all, it's hard to segment out an object from the other things in an image. Real scenes are cluttered with other objects:
* In the real world, *we can move around*, and so we have motion cues. We also have *two eyes*, so we have stereo cues. You don't get those in static images.
* So it's very hard to tell *which pieces go together* as ***parts*** of the same object.
* Also, *parts of an object* can be hidden behind *other objects*, and so, you often *don't see the whole of an object*. We humans are so good at doing vision that we don't often notice this.
* Lightning: Another thing that makes it *very hard to* ***recognize objects*** is that the intensity of a pixel is determined as much by the lighting as it is by the ***nature of the object***.
* So, for example, a black surface in bright light will give you much more intense pixels than a white surface in very gloomy light.
* Remember, to recognize an object you've got to *convert a bunch of numbers* *(i.e. the intensities of the pixels)*, into a class label.
* And, these *intensities* are *varying* for all sorts of *reasons* that have *nothing to do with the* ***nature of the object***.
* It also nothing to do with the *identity of the object*.
* Deformation: Objects can also deform in a variety of (non-affine) ways.

|  |  |
| --- | --- |
| * So for *relatively simple things* like *handwritten digits* there's a wide *variety* of *different shapes* that have the **same name**. * A "hand-written 2" for example could be very italic, or with just a cusp instead of a loop. Or it could be a very loopy 2 with a, a big loop and very round or just a cusp. * Affordances: *Object classes* are often defined by ***how they are used***. |  |

* It's also the case that for ***many types of object***, the class is defined more by *what the objects used for*, than by its *visual appearance*.
* Eg. consider chairs. ***Chairs*** are ***things*** designed for ***sitting*** on so they have a wide variety of physical shapes. There's a huge variety of things we call chairs, from armchairs to modern chairs made with steel frames and wood backs to basically anything you can sit on.
* **More things that make it hard to recognize objects**
* Viewpoints: One other thing that makes it hard to recognize objects, is that we have *different viewpoints*.
* There's a *wide variety of* ***viewpoints*** from which we can *recognize* a **3D** object.
* The *changes in* ***viewpoint*** cause changes in the images that *standard machine learning methods* cannot cope with.
* The problem is that *information hops between* the ***input dimensions*** (i.e. pixels), typically envision the *input* *dimensions* correspond to *pixels*,
  + If an object *moves in the world* and you don't move your eyes to follow it, the *information about the object* will occur on *different pixels*.
* *That's not the kind of thing* we normally have to deal with in ***machine learning***.
* Imagine a medical database in which the age of a patient sometimes hops to the input dimension that normally codes for weight!
* Just to stress that point, suppose we had a medical database in which one of the inputs is the age of a patient and another input is the weight of the patient.
* And we start doing machine learning, and then we realize that some *coder has actually changed* some inputs. For example, one of the coders put weight where they should've put age, and they put age where they should have put weight.



* To apply *machine learning* we would first want to eliminate this dimension-hopping.
* Dimension Hopping: Obviously, we wouldn't just carry on doing our learning. We'd try and do something to fix that. Because that's going to make everything go wrong. We call that phenomenon dimension hopping: when ***information*** jumps from ***one input dimension*** to ***another***.
* And that's what Viewpoint does and it's something we need to fix. And preferably we'd like to *fix it in a systematic way*.

**5.2 Achieving viewpoint invariance**

In this section we'll discuss about the issue of ***viewpoint invariance***. Each time we look at an object in the ***scene***, we typically have a ***different view point***. So the object shows up on different pixels.

This makes ***object recognition*** very unlike most ***machine learning tasks*** and we're going to talk about various ways of dealing with that issue.

* **Some ways to achieve viewpoint invariance**

A number of *different ways* have been suggested for coping with *view point variations*. We humans are so good at it that we don't really appreciate how difficult it is.

* It's one of the main difficulties in making computers *perceive* and
* There still aren't *generally accepted solutions*, either in *engineering* or in *psychology*.
* There are several ***different approaches****:*

1. The *first approach* is to use ***redundant invariant features***.
2. The *second approach* is to put a box around the object so that you can normalize the pixels.
3. The *third approach* is to use replicated features, and pool them. This is called Convolutional Neural Nets (CNN).
4. The fourth approach is to use a ***hierarchy of parts*** and to explicitly represent the *places of the parts* relative to the camera or retina.

|  |  |
| --- | --- |
| * **The invariant feature approach**   The invariant feature approach says you should ***extract a large and redundant set of features***. And they should be features that are invariant *under the transformations* like translation, rotation and scaling. |  |

* As an example of an invariant feature, consider a *pair of roughly parallel lines*, with a red dot between them.
* That's actually being suggested as the feature the baby herring gulls use for knowing what to peck for food. If you paint that feature on a piece of wood, they'll peck at the appropriate place on the piece of wood. (Herring gulls are birds, they have a red spot below their beak, that spot is crucial for the parents when *feeding* the *chicks*).
* With enough ***invariant features***, there's only one way to ***assemble*** them into an ***object*** or an ***image***.
* We *don't actually need to represent the relationships between features directly* because those relationships are captured by other features.
* This has been pointed out for strings of letters by a psychologist and pointed out in vision by Shimon Ullman.
* It's a sort of acute point that all we need is a big bag of features, because with *overlapping and redundant features*, ***one feature*** *will tell you how two* ***other features are related***.
* Unfortunately, if you're doing recognition, you're going to get a whole bunch of features that are composed of parts of different objects.
* They'll be very misleading for recognition. So we must avoid forming features from parts of different objects.
* **The judicious normalization approach**

As second approach here we discuss about judicious normalization.

* Put ***a box around the object*** and use it as a ***coordinate frame*** for a set of ***normalized pixels***.
* So if you look at that upside down capital letter **R**, (we put a box around it). Also we've ***labeled*** a 'top' and a 'front' for that box.
* Relative to that box, the R's vertical stroke at the back, and it's loop facing forwards at the top.
* So if we describe features of the R *relative to that box*, they're going to be invariant (assuming R is a rigid shape).

|  |  |
| --- | --- |
| * It solves dimension hopping: Putting a ***box around a rigid shape*** solves the *dimension hopping problem*. It gets rid of the effect of *changes in viewpoint*. * If we choose the box ***correctly***, the ***same part*** of an ***object*** would always occur on the ***same normalized pixels***. It doesn't have to be a *rectangular box*. * We can provide ***invariance*** to not only translation, rotation & scale but also things like sheer and stretch. * Unfortunately, ***choosing*** *the box is* ***difficult***. It's difficult because * we might have ***segmentation errors***. * We might have occlusion so you can't just shrink a box around things. * We might have unusual orientations. |  |

* That example of the **upside down R** makes it clear that we have to use our knowledge of: ***what the shape is*** to help us decide ***what the box is***.
* We need to recognize the shape to get the box right! For example, if we had a character that was like a lowercase D, but with an *extra stroke coming out of the loop of the D*. We would see that *as an upright one of those characters*.
* So it's a ***chicken and egg problem***. In order to ***get the box right***, we need to ***recognize the shape***. In order to ***recognize the shape***, we need to ***get the box right***.
* Many psychologists think we do mental rotation to deal with shapes that aren't oriented right. This is complete nonsense.
* That **capital letter R** you recognize perfectly *before* you do any *mental rotation*.
* Indeed, you ***need to recognize*** that it's an ***R*** and it's ***upside down***, in order to know ***how to rotate it***.
* You use mental rotation for dealing with judgments like handedness. That is, is it **a correct R** or **mirror image R**?
* You *can't tell* that *without doing mental rotation*. The *mental rotation is not used* for dealing with the fact that it's *upside down* when we want to *recognize* it.
* **The brute force normalization approach**

This approach is widely used in computer vision. Particularly for detecting upright things like faces or house numbers in *unsegmented* *images*.

The brute force normalization approach works like this: You use well segmented, upright images that you can *judiciously put a box around* when you train the recognizer. And then at test time, when you have to deal with cluttered images, you ***try all possible boxes*** in a ***whole range*** of positions and scales.

* When *training the* ***recognizer***, use well-segmented, upright images to fit the correct box.
* At test time try all possible boxes in a range of positions and scales.
* This approach is widely used for detecting upright things like faces and house numbers in unsegmented images.
* It is much more efficient *if the recognizer* can cope with some ***variation in position*** and ***scale*** so that we can use a coarse grid when trying *all possible boxes*.

**5.3 Convolutional nets for Digit Recognition**

Now we're going to discuss about Convolusional Neural Networks (CNN) for handwritten digit recognition. It was one of the big success stories of neural networks in the 1980s.

* The Deep Convolutional Nets developed by Yann LeCun and his *collaborators* did a really good job of ***recognizing handwriting*** and were actually *used in practice*.
* It's one of the few examples from *that period* of Deep Neural Nets that it was possible to *train on computers that existed then*, and that *performed* really *well*.
* **The replicated feature approach (currently the dominant approach for neural networks)**

*Convolutional Neural Networks (CNN)* are based on the idea of replicated features.

* We know ***objects move*** *around* and show up on ***different pixels***, if we have a feature detector that's useful in *one place in the image*, it's likely that the same feature detector will be useful *somewhere else*.
* The idea is to build ***many different copies*** of the same ***feature detector*** in all the ***different positions***.

|  |  |
| --- | --- |
| * If you look on the right I've shown you ***three feature detectors***, which are replicas of each other. Each of them has weights to nine pixels. And those weights are identical between the three different feature detectors. * So the red arrow has the same weight on it for all three feature detectors. * And when we learn, we keep those **red arrows** all having the **same weight** as each other and we keep the **green arrows** having all the **same** **weight** as each other. * Even though the **red** and **green arrows** will have different weights. * Replicate features across scale and orientation: We could also try replicating across scale and orientation but that's much more difficult and expensive and probably not a good idea. * Replication reduces free parameters to be learned: Replication across position greatly *reduces* the number of free parameters that you ***have to learn***. So those *27 pixels* that you see in those *three replicated detectors* only have ***nine*** different ***weights***. |  |

* Use several different feature types: We don't use only **one feature type**. So we're going to have **many maps**. Each map will have replicas of the same feature, features that are constrained to be ***identical*** in ***different places***.
* So, we use *several different feature types*, each with *its own map* of *replicated detectors*.
* And then ***different maps*** will ***learn*** to ***detect different features***.
* This allows *each patch of the image* to be represented by *features* *of* *many different types*. Each ***patch*** of image will be *represented* in ***several ways***.
* **Backpropagation with weight constraints**

Replicated features fit-in nicely with backpropagation (i.e. it's easy to *learn* using *backpropagation*).

* In fact it's easy to modify the ***Backpropagation Algorithm*** incorporate any *linear constraint* between the *weights*.
* So what we do is- we compute the gradients as usual. But then we modify the gradients, so that if the ***weight satisfied*** the *linear* *constraint* **before** the **weight update**, they'll *also satisfy* the *linear constraint* **after** the **weight update**.

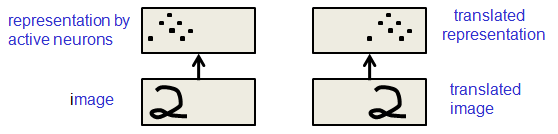
|  |  |
| --- | --- |
| * The simplest example is we want two weights to be equal. * We want . That would be true if we start off with . And we make sure that the change in , is always equal to the change in , (i.e. ). * The way we do that is- we compute the gradient of the error E with respect to and . And then we use the sum or average of those two gradients for both and . |  |

By using weight constraints like that, we can force backpropagation to learn ***Replicated Feature Detectors***.

* **What does replicating the feature detectors achieve?**

There's quite a lot of confusion in the literature about what replicated feature detectors are actually achieving.

* Many people claim they're achieving Translation Invariance. And that's not true.
* Well, at least it's not true in the ***activities*** *of the* ***neurons***.
* So if you look at the activities, what replication features achieve is Equivariance not invariance.
* Following example should make that clear.



* Here's an image, and the black dots are the activated neurons.
* Also notice the ***translated image***. And notice the black dots have also translated.
* While the *image changed*, the representation also *changed* by just *as much as the image*. That's equivariance not invariance.
* Also there is something invariant, and that's the knowledge. So if you learn Replicative Feature Detectors, if you know how to detect a feature in one place, you'll know how to detect that same feature in another place.
* So it's *important to note* that we're achieving "equivariance in the activities" and "invariance in the weights".
* Equivariant activities: *Replicated features* do not make the *neural activities* ***invariant*** to translation. The activities are ***equivariant***.
* Invariant knowledge: If a ***feature*** is *useful* in some locations during *training*, ***detectors*** for that *feature* will be available in ***all*** ***locations*** during *testing*.

* **Pooling the outputs of replicated feature detectors**

If you want to achieve some invariance in the activities, what you need to do is pool the outputs ***of replicative feature detectors***.

* We can get a *small amount* of ***translational invariance*** at *each level* of a deep net, by averaging *four neighboring replicated detectors* to give a **single output** to the **next level**.
* One advantage of this is that: it reduces the **number of** inputs to the ***next layer*** of ***feature extraction***. So that we can have many more ***different feature maps***, allowing us to learn more *different kinds of features* in the *next layer*.
* ***Maximum of the four is slightly better:*** It actually works slightly better to take the maximum of *four neighboring feature detectors*, rather than an average, but there is a problem.
* Issue after several levels of pooling: The problem is that after doing *several levels* of this kind of pooling, we've ***lost precise information*** about where things are. We have lost information about the precise positions of things.
* That's okay if we just want to recognize that it's a face.
* The fact that we've got a few eyes, and a nose, and a mouth floating about in *vaguely the* ***same position*** is very good evidence that it's a face.
* But if you want to recognize whose face it is, you need to use the *precise* ***spatial relationships*** between the eyes, the nose and the mouth.

And that's been lost by these ***Convolutional Neural Nets***. This makes it *impossible to use* the ***precise spatial relationships*** between ***high-level parts*** for ***recognition***.

* **Le Net**

The first impressive example of a ***Convolutional Neural Net*** was done by Yann LeCun and his collaborators who developed a really good recognizer for handwritten digits by using ***backpropagation*** in a ***feedforward net*** with:

* Many hidden layers.
* Many maps of replicated units in *each layer*.
* Pooling of the outputs of nearby replicated units: It had pooling *between layers*. So you ***pool*** *adjacent replicated units* before you send them to the ***next layer***.
* But it also used a wide net that could ***cope with several characters*** at once. And that would work even if the ***characters overlapped***.
* So you *didn't have to segment out* ***individual characters*** before you *fed* them to their *net*.
* A clever way of training a complete system, not just a recognizer: They used a clever way of *training* a complete system. They *weren't just training* a ***recognizer*** *of* ***individual characters***.
* They were *training a complete system*, so that you put in ***pixels*** *at one end* and you get out ***whole zip codes*** at the *other end*.
* They used Maximum Margin: In training that system they used a method that would now be called maximum margin. But when they did it, it was *way before* maximum margin had been invented.
* This net was used for reading ~10% of the checks in North America. So it was of great practical value.
* There were some impressive demos of LENET at, on Yann's webpage <https://yann.lecun.com>. You should really go look at all of them. Because they show you just how well it copes with variations in *size*, *orientation*, *position*, *overlap of digits*, and all sorts of *background noise* that would kill most methods.
* **The architecture of LeNet5**

The architecture of LeNet-5 looks like following.

* There's an input, which is pixels ().
* And then there's a whole sequence of feature maps followed by sub sampling.
* C1: feature map: In the *C1: feature map*, there are *six different maps*. Each of those maps has of those maps that contain small features that just look at I think  **pixels**. And their *weights* are *constrained* together.
* So per map there's only about *9 parameters*. That makes learning much more efficient. It means you need much less data.

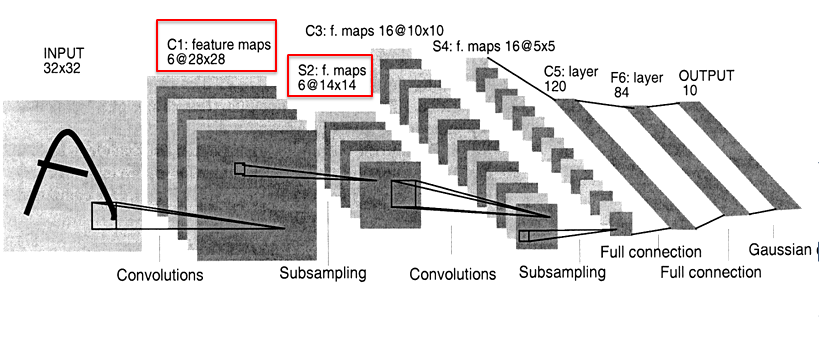


Fig: The architecture of LeNet5

* sub-sampling (pooling): After the feature map, there's what they call sub-sampling which is now called pooling.
* Here you ***pool together*** the ***outputs*** of a bunch of ***neighboring replicated features*** in C1.
* This pooling gives you a smaller map, which will then *provide the* ***input*** to the *next layer*, which is *discovering* more ***complicated replicated features***.

As you go up this ***hierarchy***, you get *features that are much more complicated*, but are *more* ***invariant*** *to* ***position***.

* **The 82 errors made by LeNet5**

Here's the errors that LeNet5 made. There are *10,000 test cases*, and these are the *82 errors* that LeNet5 makes. So it's doing better than 99% correct.

* Notice that most of the errors are cases that people find quite easy to recognize.
* The human error rate is probably 20 to 30 errors but nobody has had the patience to measure it.



Fig: The 82 errors made by LeNet5

* Of course there might be digits that LeNet5 got right and you would get wrong. So you have to be ***careful*** in ***estimating*** the ***error rate***.
* You can't just look at these 82-errors and ask which ones you'll get right and which ones you'll get wrong. You have to worry about all those other ones that LeNet5 might've got right and you might've got wrong.
* **Priors and Prejudice**

***Inject prior knowledge in machine learning:*** Now we discuss about how to ***inject prior knowledge*** in ***machine learning***, using ***Neural Networks***.

* We can put in prior knowledge as it is done LeNet5, by the design of the network. We can put our *prior* ***knowledge*** *about the* ***task*** into the network by designing appropriate:
* ***Connectivity:*** We can have local connectivity.
* ***Weight constraints:*** We can have weight constraints.
* ***Neuron activation functions:*** Or we can choose neural-activities that are particularly appropriate for the task we're doing.
* This is ***less intrusive*** than hand-designing the features.
* But it still prejudices the network towards the *particular way of solving the problem* that we had in *mind*. We have an idea about how to do object recognition by gradually making bigger and bigger features. And by ***replicating*** these ***features*** across ***space***. And we ***force the network*** to do it that way.
* There is an alternative way to put in prior knowledge that gives the network a *much freer hand*.
* We can use our prior knowledge to get or create a whole lot more training data.
* One of the first examples of this was work by Hofmann and Tresp (Hofman & Tresp, 1993) on trying to model what happens in a steel mill. They wanted to know the relationship between what *comes out of the steel mill* and various *input variables*, and they actually had an, big old *Fortran-simulator* that would allow them to ***simulate*** the ***steel mill***.
* Of course, the *simulator wasn't reality*. It was making all sorts of *approximations*.
* So they had real data, and also a simulator. Then they run the simulator in order to create some synthetic data.
* Then the created synthetic data added to the real data.
* Hofman & Tresp showed that it could do better than just using the real data alone.

However, big Fortran simulator was only worth a few dozen extra real examples, but nevertheless, they *made the point*.

* Of course, if you generate a lot of synthetic data, it may make learning take much longer.
* So in terms of the speed of learning, it's much more efficient to put in knowledge by using things like: connectivity and weight constraints, as was done in LeNet5.
* But as computers get faster, this other way of putting in knowledge, by generating Synthetic Examples, begins to look better and better.
* In particular, it allows optimization to discover clever ways of using the multilayer network that we didn't think of. If fact, we might never fully understand how it does it.
* If we just want good solutions to a problem, that might be fine.
* **The "Brute Force" approach**

Using the *idea of* ***synthetic data***, there's a ***brute force approach*** to handwritten digit recognition.

* ***Lenet5*** uses knowledge about invariances to design:
* The local connectivity
* The weight-sharing and
* The pooling.

And this achieves about 80 errors. This can be reduced to about 40 errors by using many different ***transformations*** *of the* ***input*** and other tricks including ***synthetic data***, Ranzato (Ranzato 2008)was able to get that down to about 40 errors.

* A group in Switzerland, led by Ciresan et. al. (2010) went to down with injecting knowledge by ***putting*** in ***synthetic data***.
* They injected the ***knowledge*** *of* ***invariances*** by creating a huge amount of carefully designed extra training data (instructive synthetic data).
* For each training image, they produce many *new training examples* by applying many *different transformations*. So for every real training case, they transformed it to make many more training examples.
* They can then train a large, deep, dumb net on a GPU without much *overfitting*. They achieve about 35 errors.
* They trained a ***large net*** with ***many units*** per ***layer***, many ***layers*** on a graphic processor unit (GPU).
* The GPU gave them a *factor of thirteen computation*. And because of all the *synthetic data* they put in, it didn't *overfit*.
* If they just use a *large net* with a *GPU*. It would have been a disaster that *over fitted terribly* that they would have *done* on the *training data* but *terribly* on the *test data*.
* So they were really combining three tricks.
* Put your effort in to ***generating*** lots of ***synthetic data*** then
* ***train*** a large ***dumb net*** on a ***GPU***.
* ***Reduce overfitting*** to achieve 35 errors.

|  |  |
| --- | --- |
| * **The errors made by the Ciresan et. al. net**   Following are the 35 errors made by the **Ciresan et. al. net**. The *top printed digit* is the ***right answer***. And the *bottom two printed digits* are best two guesses by the net.   * The right answer is almost always in the top 2 guesses: Notice that the Net is nearly always get the right answer in their best two guesses. There are only five cases where the Net failed. * With model averaging they can now get about 25 errors: With some more work by building *several different models like this* and then using a consensus to decide *what the digit was*, they managed to get down to about **25** errors. That is very near to the *Human Error Rate*. |  |

* **How to detect a "significant drop" in the "error rate"**

One question this work raises is: how do you tell if a *model makes 30 errors* is really *better* than a *model that makes 40 errors*? Is that significantly different? Is 30 errors in 10,000 test cases significantly better than 40 errors?

* *It all depends on the particular errors!* Rather surprisingly, it depends on which errors they make. The *numbers* ***don't provide*** you *enough* *information*. You *have to know* *which ones* they get ***wrong*** and *which ones* they get ***right***.
* McNemar test: This kind of statistical test is called the McNemar test that uses the particular errors and is far *more sensitive* than just using the *numbers*.
* Example: The ***McNemar*** test uses the ***particular errors*** and can be much more ***powerful*** than a test that just uses the ***number of*** ***errors***.
* Consider following example of table (confusion matrix). It shows you, in the *top left hand corner*, how many examples ***Model-1*** and also ***Model-2*** got wrong, and it is **29**.
* And in the *bottom right*, it shows you how many examples ***Model-1*** and also ***Model-2*** got right, and it is **9959**.
* In the McNemar test, you can just ignore those *numbers in* ***black***.

|  |  |
| --- | --- |
| * We're only interested in the cases where ***Model-1*** got right but ***Model-2*** got wrong, or ***Model-2*** got right and ***Model-1*** got wrong (numbers in red). * And if you look at that, there's **11:1** ratio, and it turns out that's pretty significant. ***Model-2*** is definitely better than ***Model-1***. That didn't happen by accident, almost certainly. |  |
| * By contrast if you look at that table, again. * ***Model-1*** is making 40 errors, ***Model-2*** is making 30 errors, but now ***Model-1*** is *winning 15 times* when ***Model-2*** *loses* and ***Model-2*** is *winning 25 times* when ***Model-1*** *loses*. There is now **25:15** ratio. * Now, these difference is *not very significant* so we *wouldn't be confident* that ***Model-2*** is better than ***Model-1***. |  |

**5.4 Convolutional nets for Object Recognition**

There is a question that, the kinds of nets developed for recognizing handwritten digits could actually be scaled up for a real task (i. e. *recognizing objects* in high resolution color images when the scene is *cluttered*)? For that you have to do things like:

* Segmentation,
* You have to deal with 3D viewpoint,
* You have to deal with many different objects surrounding that you're not quite sure which is the intended one, and so on.

In this section we'll discuss the network developed by Alex Krizhevsky and we'll see that it is good at object recognition. Now it has been benchmarked against the best computer vision systems.

* **From hand-written digits to 3-D objects**

People worked on MNIST for many years, *gradually improving* their ability of these *networks* to *recognize handwritten digits*.

Many computer vision researchers thought this was a waste of time if you wanted to be able to *recognize real objects* in color images, because they thought *'the lessons learned from MNIST would not generalize to that domain'*.

* Following are a number of reasons why it's a much more difficult task.
* Recognizing ***real objects*** in *color photographs* downloaded from the web is much more ***complicated*** than *recognizing* ***hand-written digits***:
* Hundred times as many classes (1000 vs 10): First of all, there are many different kinds of objects. Even if we only recognize a thousand classes, that's still a factor of a hundred.
* Hundred times as many **pixels** (256 x 256 **color** vs 28 x 28 **gray**): Secondly there is many more pixels even if we use down-sampled images that are only with color pixels that's still 100 or 300 times of many pixels compared to images of hand-written digits of gray pixels.
* 2D image of 3D scene: Another factor is, you've got a **2D** image of a **3D** reality, so a lot of information is being lost.
* Cluttered scenes requiring segmentation: Any real scenes have clutter of a kind that *doesn't occur* in *handwriting*.
* In handwriting you can have overlapping letters and that requires segmentation but you don't have things like ***occlusion*** *of large parts of objects* by opaque other objects.
* Multiple objects in each image: You don't have many different kinds of objects in the same scene. And you don't have a little lighting variations that you get in real scenes.
* Now the question is: Will the same type of ***Convolutional Neural Network*** that proved to be so good on recognizing hand-written digits work for real color images?
* In the domain of ***real color images*** we probably do need to wire in some prior knowledge.
* Because, if we try and do it in the *sera san way* with *no knowledge wired in*, putting in all the knowledge by *generating* extra *training* *examples*. The *computational problem* is still *too large* for *current computers*.

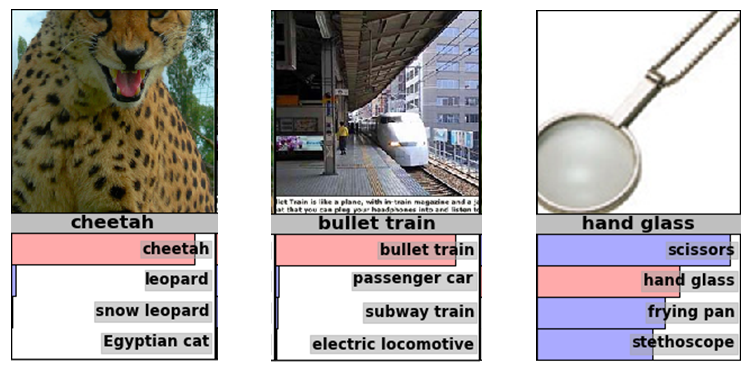
|  |  |
| --- | --- |
| * MNIST: The ***MNIST Database*** *(Modified National Institute of Standards and Technology database)* is a large database of handwritten digits that is commonly used for *training various image processing systems*. The database is also widely used for training and testing in the field of ***Machine Learning***. |  |

* **The ILSVRC-2012 competition on ImageNet**

Following competition was on a database called ImageNet. ImageNet has *more than a million images* but a ***subset*** of ***1.2 million*** was chosen.

* The **classification task** was to correctly label those *1.2 million high-resolution training images*.
* The CLASSIFICATION task: Get the **“correct” class** in your **top 5 bets**. There are **1000 classes**.
* The images were hand-labeled with a thousand different classes but this wasn't very reliable.
* There could be an ***image*** that has ***two*** of those thousand different ***objects in it*** and only ***one*** of them is ***labeled***.
* So, to make the task feasible, the *computer vision system* is allowed to make **five bets**.
* To get it right, one of those bets need to corresponds to the label that a person has given the image.
* The LOCALIZATION task: For each bet, put a **box** around the **object**. Your box must have at least **50% overlap** with the **correct box**.
* For the ***localization task*** you have to place a **box around an object** once you've *recognized it* and ***to get it right*** your **box must have** at least a **50% overlap** with the **correct box**.
* The ***reason*** for the ***localization task*** is that *many computer vision systems* use a Bag of Features approach.
* For the *whole image* or a *quadrant of the image* they know ***what*** the ***features*** are, but they ***don't know where they are***. This allows them to *recognize objects* but without knowing exactly *where they are*.
* That's very unlike how people behave, except people with a *curious kind of brain damage* called ***balance syndrome*** where they can *recognize objects* and *not be sure where they are*.
* On this task, people tried some of the ***best*** existing ***computer vision methods***. So, leading groups from *Oxford* and the *French National* *Research Labs: INRIA* and *Xerox's European Research Center: XRCE* and various other universities tried this task and discovered it's very hard.
* The **computer vision systems** typically used complicated **multi-stage systems**.
* The **early stages** of these **systems** are typically **hand-tuned** by *optimizing* a *few parameters* using some of the data.
* The top stage of these systems is always a learning algorithm.
* But they *don't learn all the way through* (the way that a deep neural net does when its trained to do back propagation).
* They don't have end-to-end learning, where the *parameters* used in the *early feature detectors* are being *influenced* by how useful they are for making *final decision about classes*.
* **Examples from the test set (with the network’s guesses)**

So here are some examples from the test-set to show you what the data is like. We already saw some examples in previous sections, but here's some more.



* Case 1: We can see there is a Cheetah in the first image but lot of its body missing. It doesn't have ears, legs or paws.
* The predictions are the ***un-normalized probabilities*** of Alex Krizhevsky deep-neural-network. And you can see it's 95% confident that, that is a Cheetah, 2nd guess is a leopard. It also understands there's other possibilities, like a snow leopard, though it has the wrong color for a snow leopard, or an Egyptian cat.
* Case 2: In the 2nd image there are *many objects* in the *image* and the ***object of interest*** has only a very *small fraction of the pixels*.
* The network 90% correctly says Bullet Train. But it also has other bets, like Subway Train or Electric Locomotive, which are presentable bets.
* If you look at the image, there's lots of other things that could be labeled, like the **roof** which occupies a much larger fraction of the image *than* the *train* or the *pillar* that's supporting the roof or other things like the *large apartment block* in the background.
* In these kinds of images you really have to be able cope with the fact that there are lots of alternative targets.
* Case 3: The last image shows a different kind of example where *there is no background clutter*. The ***object*** is quite well ***isolated***. And the *network* *doesn't get it right* for its ***first bet***, but it does get it in its ***top five bets***.
* Here the network ***isn't******confident*** *about* ***anything***. These are the ***relative probabilities***, and the network correctly *realizes* *it doesn't* really *know*.
* And if you look at the other possibilities, they're all perfectly plausible.
* **Error rates on the "ILSVRC-2012 competition"**

So how did the systems do on this (above example) data? Following are the Error Rates for the different ***Computer Vision Systems***.

* Notice that the best systems are all very similar. Following rates are the best system from each group.

|  |  |  |
| --- | --- | --- |
|  | Classification | Classification & localization |
| University of Tokyo | **26.1%** | **53.6%** |
| Oxford University Computer Vision Group (possibly the best group in Europe) | **26.9%** | **50.0%** |
| INRIA (French national research institute in CS) + XRCE (Xerox Research Center Europe) | **27%** |  |
| University of Amsterdam | **29.5%** |  |
| **University of Toronto (Alex Krizhevsky)** | **16.4%** | **34.1%** |

* So you'll guess from this that: it is going to be ***hard*** to be ***26%***, and if you do ***beat 26%*** you're comparable with the very ***best*** ***computer vision systems***.
* Notice Alex Krizhevsky's neural net got **16.4%**error. It's a huge gap. Normally, in these competitions you don't see big gaps like that.
* **A neural network for ImageNet**

**Alex Krizhevsky's** network (NIPS 2012) works like below. It's a *very Deep Convolutional Neural Net* of the type pioneered by Yann LeCun. This kind of network first used for digit recognition and then Yann later applied it to recognizing real objects.

* Currently we're using all the lessons that we learned by Yann's group and from Yoshua Bengio's group and various other groups, developing these ***Deep Neural Nets*** for doing real vision.
* Architecture:

1. 7 hidden layers: It has **7 hidden layers**, which is deeper than usual and that's not counting some of ***the max-pooling layers***.
2. Early layers were convolutional: The early layers are convolutional. We could probably get away with using just **local receptive fields**, without tying any weights, if we had a much bigger computer.

* But by making them *convolutionary*, you *cut* down the *parameters* a lot, so you *cut down* the amount of *training data* (you need a lot of data) which *cuts* down the amount of *computation time* a lot.

1. Last two layers were globally connected: The last two layers were globally connected and that's where most of the parameters are. There are about **sixteen million** parameters ***between each pair*** of those ***layers***.

* What the last two layers are doing is: looking for ***combinations of local features*** that were ***extracted*** by the ***early******layers***. And obviously there are many combinations to look for, and that's why you need a lot of parameters there.
* The activation functions were:

1. ReLU in each hidden layer: Rectified linear units in every hidden layer. These train much faster and are more expressive than logistic units.

* Most of the people seriously applying **Deep NN** to *real images* to the *object recognition* of using nice swish ReLU.

1. Competitive Normalization: Competitive Normalization were used within a layer. It is used to ***suppress*** the ***hidden activities*** of a ***unit*** when nearby units have stronger activities (other units that are looking nearby localities are very active).

* This helps a lot with variations in intensity.
* For example, you might have an edge detector, which gets somewhat ***active*** due to some ***fairly faint edge***. And that's pretty much ***irrelevant***, if there are much ***more intense things*** around.
* **Tricks that significantly improve generalization**

There are other tricks that we used to significantly ***improve*** the generalization of Alex's net.

* Enhancing the data: First of all, we use the trick of enhancing the data by using transformations. We took down-sampled images from the competition. But *instead of using* those *whole images* Alex Krizhevsky took random patches from those images. That gave him more images to train on and helped him deal with translation invariance. Even though they're convolutional nets, and that's still a help.
* Train on random patches from the images to get more data. (Taking px portions from px image).
* Also use left-right reflections of the images. He also used ***left-right reflections*** of the images, which again ***doubled*** the amount of data.
* He didn't use ***up-down*** reflections. Because, gravity is very important.
* ***Left-right*** reflections don't really change what things look like much unless they're things like writing (alphabet, symbol etc).
* At test time, he *doesn't just use one patch*. He uses a number of ***different patches***, the four 224×224 corner patches, the 224×224 middle patch, that gives him five-patches, and then the ***left right-reflections*** of all those, that gives him ***ten-patches***.
* Also at test time, he runs all *ten-patches through the network* and then *combines* the *opinions* from *ten different patches*. The four 224x224 corner patches plus the central 224x224 patch plus the reflections of those five patches.
* Drop-out- regularization: In the top layers, where most of the parameters are, he used a new regularization technique, called Drop-Out, which is very effective. This ***Drop-out- regularization*** stops the network over fitting. That's worth several percent in his results.
* Use “Dropout” to regularize the weights in the ***globally connected layers*** (which contain most of the parameters).
* Dropout means that half of the hidden units in a layer are randomly removed for each training example. This *stops* ***hidden units*** from relying too much on ***other hidden units***.
* They can't learn to *fix up the errors* left over by the *other hidden units* in that layer, because the *other hidden units* might *not be there* no matter be *fixing up an error* that *doesn't exist*.
* So the ommited-units have to become more individualist. They have to *individually do useful things,* but they still have to do ***useful things*** that are ***different*** from what the ***other survivors*** do.
* Therefore drop-out is stopping too much cooperation between the hidden units. Since a ***lot of cooperation*** is very good for fitting the training data. But if the ***test distribution*** is significantly ***different***, then all that *cooperation* causes ***over-fitting***.
* **The hardware required for Alex’s net**

Hardware upgrade: Alex couldn't have done this work without significant hardware, but the hardware only costs a few thousand dollars now. Alex is a very good programmer, and he used a very efficient ***implementation*** of ***Convolutional Neural Nets*** on two Nvidia GTX 580 graphics processors (over 1000 fast little cores).

* Each of those GPU has over **500** fast little cores, which are very good at doing arithmetic and not much good at anything else.
* The GPUs are very good at doing matrix-matrix multiplies.
* For example, if you stack together the vector of activities of a *hidden layer*, over *many training cases*, that gives you a matrix. And now you multiply that by matrix of weights to figure out the activities in the next hidden layer for all *those training cases*, and if both those matrices are big, the GPU's give you a huge advantage.
* They give you about a factor of 30.
* GPUs have very high bandwidth to memory (ram), and that's needed for ***Neural Nets***. Cause in Neural Nets you keep wanting to know another **weight** so that you can **multiply** it by an **activity**.
* There's millions of these weights, so you can't keep them all in the cache.
* Using all that hardware, he could ***train*** his final ***network***, in a ***week***.
* It can also combine results from ***10 different patches*** at ***Test-Time*** very quickly. You can run it at just about the frame rate.
* In the future we are going to be able to spread this kind of network over a **large number** of **cores**. People at Google are already experimenting with that.
* We can ***spread a network*** over ***many cores*** if we can communicate the ***states*** fast enough.
* Google has *already simulated networks* with ***1.7 billion connections*** and I think that it's only going to get bigger.
* As cores get cheaper and datasets get bigger, ***big deep neural*** nets will ***improve faster than old-fashioned*** (i.e. pre Oct 2012) computer vision systems.
* Because, they *don't involve* much ***hand engineering***, and they can make very good use of *huge data-sets* and *huge* amounts of *computation*.
* Now on all the best object recognition systems, at least of static images, will use big deep neural nets.
* **Finding roads in high-resolution images**
* **Vlad Mnih (ICML 2012)** used a non-convolutional net with local fields and multiple layers of ReLU to find roads in cluttered aerial images.
* He used a net ***with local fields*** but ***without convolution*** to extract roads from aerial images. These are *cluttered aerial images* of *urban scenes*.
* He uses multiple layers of ReLU.
* It takes a large image patch and predicts a binary road label for the central **16x16** pixels (whether each of those pixels is a piece of road or not a piece of road).
* There is *lots of labeled training data* available for this *task*. That's because maps tell you where the centre lines of roads are and roads are roughly fixed width.
* So from the *vectors* in the map that tell you where the *centre line of the road is* you can estimate which pixels are probably road.
* The task is hard for many reasons, following are the normal kind of ***vision problems***:
* Occlusion by buildings, trees and cars- sitting on the road. Also roads are occluded by buildings because a plane *isn't looking* ***straight down*** when it takes the photograph.
* Shadows, Lighting changes: The shadow effects from building, the major lighting changes depending on whether it's a **sunny day** or a **cloudy day** for example.
* Minor viewpoint changes: The plane is basically looking downwards, but in any *large photo* it can't be looking *straight downwards* at *every pixel*.
* The worst problems in this data are incorrect labels:
* *Badly registered* maps:
* You get incorrect labels because the maps *aren't perfectly registered*. For most purposes, you don't need a map to be registered better than a few meters. The pixels are about one meter square in this data.
* So if the registration of the map is off by three meters, you're going to get *at least three of the labels wrong for pixels*, across every road.
* Arbitrary decisions about what counts as a road.
* Another, severe problem, is that the *people making maps* have to *make arbitrary decisions* about what counts as a road and what counts as a laneway.
* So in a map, you've no idea whether that's gonna be considered to be a road or a lane-way. You simply don't know what label it's gonna get from the map.
* Big neural nets trained on big image patches with millions of examples are the only hope.

* **The best road-finder on the planet?**
* So, following is what the data looks like. This is a part of Toronto. There is two patches extracted from that image.
* If you look at those patches, you can see it's not trivial to tell which the road pixels are. On the right, is the output of **Vlad Mnih's** system.

|  |  |
| --- | --- |
|  |  |

* Green is correctly identified pixels of road, and red means things that his system thought might be road, but actually aren't.
* Actually that thing is a parking lot but you can see why he might have thought it was a road.