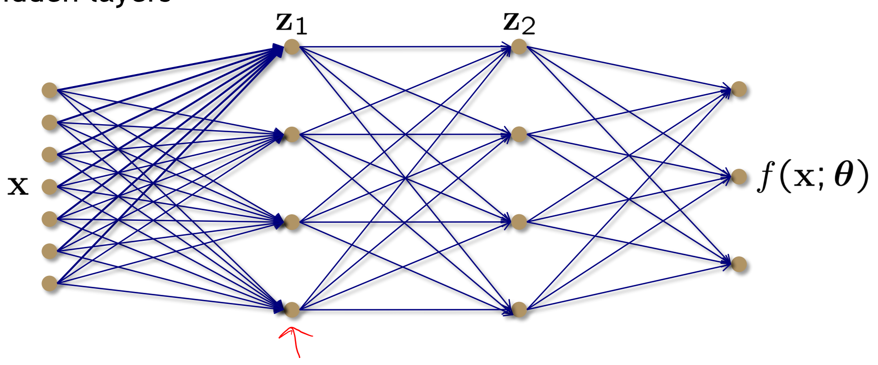
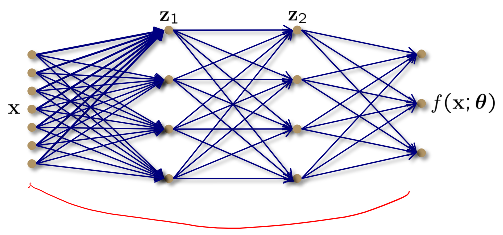
## Multi-layer neural networks

The neural net framework, along with the backpropagation algorithm, can easily be extended to networks with multiple hidden layers



Has had a resurgence in recent years under the name ***"deep learning"***

## Deep learning



## Why *deep* learning?

Because it works!

Maybe a better question is ***why not*** deep learning?

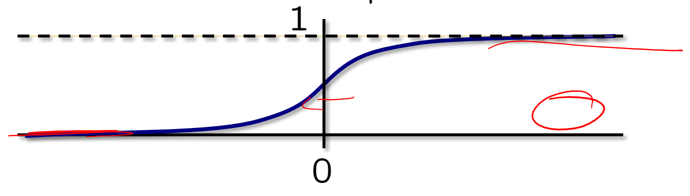
- there are ***lots*** of parameters to optimize over
- this means that ***overfitting*** is a real risk
- success requires careful design of your network structure and use of ***regularization*** and other clever techniques
- success may also require ***massive*** amounts of data
- these days, maybe it is not always so hard to get massive amounts of data, but this makes training ***slow***
- getting success on big problems in a reasonable amount of time requires careful initialization and the use of many tools from optimization

## Deep learning architectures

With these challenges in mind, what choices do people typically make in practice?

- choose activation functions and a loss function that make optimization easier
- want to avoid situations where the gradient is tiny, but you are far from the true solution
  - "saturation"

$$g(t) = \frac{1}{1+e^{-t}}$$



## Choice of loss function

***Regression***

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{n} \|\mathbf{y}_i - f(\mathbf{x}_i;\boldsymbol{\theta})\|_2^2$$

***Classification***

$$L(\boldsymbol{\theta}) = -\sum_{i=1}^{n} \mathbf{y}_i^T \log \underline{f(\mathbf{x}_i;\boldsymbol{\theta})}$$

cross entropy

## Choice of output units

The choice of $h$ (mapping from the last hidden layer to the output) depends on the application

***Regression***

$$h(t_k) = t_k \qquad t_k = \mathbf{w}_k^T\mathbf{z}$$

***Classification***

$$h(t_k) = \frac{e^{t_k}}{\boxed{\sum_{j=1}^{K} e^{t_j}}} \qquad t_k = \mathbf{w}_k^T\mathbf{z}$$

## Saturation

Notice that the combination of loss function and output unit work together to avoid the saturation problem

***Regression***

$$L(\boldsymbol{\theta}) = \sum_{i=1}^{n} \|\mathbf{y}_i - \mathbf{W}\mathbf{z}_i\|_2^2$$

***Classification***

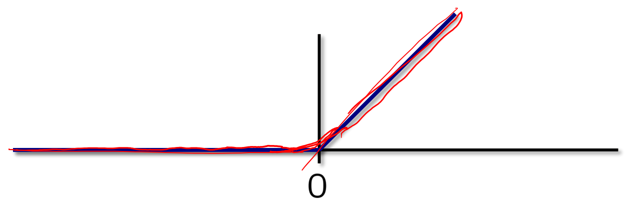
$$h(t_k) = \frac{e^{t_k}}{\sum_{j=1}^{K} e^{t_j}} \qquad L(\boldsymbol{\theta}) = -\sum_{i=1}^{n} \mathbf{y}_i^T \underline{\log h(\mathbf{W}\mathbf{z}_i)}$$

One can show that this combination will only "saturate" when $h(\mathbf{W}\underline{\mathbf{z}_i})$ is already predicting $\mathbf{y}_i$ correctly

## Choice of hidden units

For hidden units in a deep architecture the most common choice is the ***rectified linear unit (ReLU)***

ReLUs use the activation function $g(t) = \max\{0, t\}$



Note that the derivative is (usually) very easy to calculate

Not differentiable at zero, but generalizations exist

## Regularization

To avoid overfitting the data, regularization is crucial

Choose $\boldsymbol{\theta}$ to minimize

$$L(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|_2^2 \quad \text{or} \quad L(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|_1$$

This encourages small (or zero) weights

Note that when the weights are small, the effect of the nonlinear functions $g, h$ can go away

If we constrain all weights to be small, the entire network can approximately reduce to a simple linear classifier

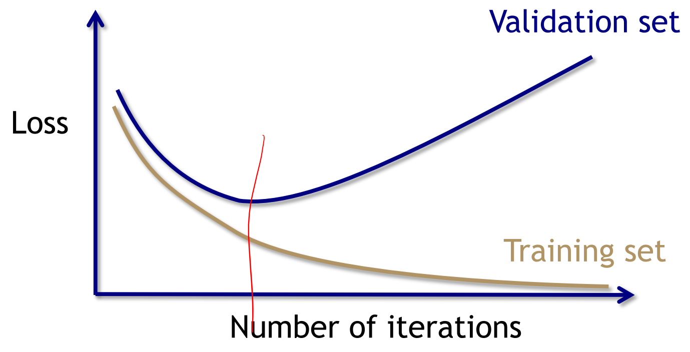
Encouraging small weights in the network reduces the effective number of degrees of freedom

## How many layers/units?

- Choosing the number of hidden layers and number of units in each layer is more art than science
- In general, it seems to be better to have too many parameters rather than too few, and rely on regularization to avoid overfitting
- Additional strategies to control overfitting include
  - early stopping
  - dropout
  - parameter sharing
  - dataset augmentation

## Early stopping

When running gradient descent, you would think that more iterations would always be better…



Early stopping serves as a form of regularization

## Dropout

Another way to control overfitting is through a technique called ***dropout***

At each step of gradient descent, a randomly selected subset of the weights are “masked” (temporarily set to zero)

A gradient is computed, a step is taken, and the process is repeated with a new subset of weights being masked

This can be thought of as approximating a method of ***bagging*** which trains many neural networks and averages the output, without actually having to train an ensemble of networks

Also can be show to act as a form of regularization

## Parameter sharing

Instead of using regularization to keep the parameters small, another common strategy is to place constraints on the parameters

Force sets of parameters to be equal

This is the underlying idea of ***convolutional neural networks***, where the linear mapping to the hidden layers is of the form of a convolution

Weights can be modeled via circulant matrices (same parameters for each row)

More on this later…

## Dataset augmentation

Overfitting is fundamentally a problem of having too many parameters and not enough data

In many application areas, it can be possible to generate more training data on demand

For example, images can be

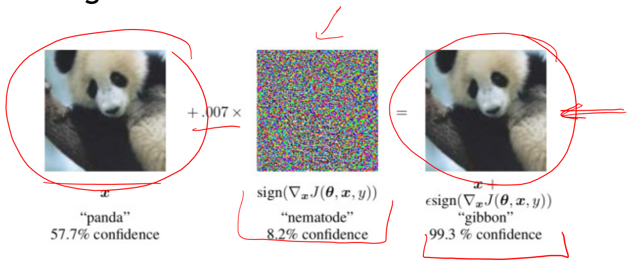
- translated
- rotated
- zoomed
- warped, occluded, and subjected to other distortions

without changing the correct class label

It may be possible to generate lots of extra training data…

## Adversarial training

If you work at it, it is also possible to generate some rather strange training data…


$+ .007 \times$
$=$
$\boldsymbol{x}$
"panda"
57.7% confidence
$\mathrm{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence
$\boldsymbol{x} + \epsilon\mathrm{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

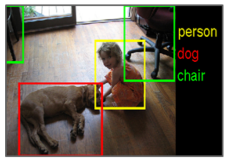
This highlights some strange behavior, but also provides a way to generate augmented data that can help make deep networks much more robust

## Large-scale gradient descent

Some of the most impressive results achieved by deep architectures have been on large-scale image datasets

Consider the ImageNet dataset

- 14 million images
- 1000 classes



Each gradient step is incredibly expensive…

- make each step faster
- try to take fewer steps
- initialization is super important!
  - careful normalization if using random weights
  - pre-training using unsupervised learning techniques

## Stochastic gradient descent

The best way to make gradient descent faster on large data sets is something we have already seen…

***Stochastic gradient descent***

Simply compute an estimate of the gradient at each iteration by randomly sampling a subset of the training data instead of working with the whole dataset

This might result in an increase in the total number of iterations required to converge, but the speedup in each iteration can be so large that this tradeoff is often worth it

## Other first-order methods

We have talked a lot about gradient descent, but there has been a lot of research recently in other first-order variants that can often result in much faster convergence

***Heavy ball method***

(Gradient descent with momentum)

$$\boldsymbol{\theta}^{(r+1)} = \underbrace{\boldsymbol{\theta}^{(r)} - \alpha_r \nabla L(\boldsymbol{\theta}^{(r)})}_{\text{gradient update}} + \underbrace{\beta_r(\boldsymbol{\theta}^{(r)} - \boldsymbol{\theta}^{(r-1)})}_{\text{momentum}}$$

Momentum term can cancel out "oscillations" in certain dimensions, resulting in convergence in fewer iterations

See also ***Nesterov's accelerated methods***

## Convolutional neural networks

The neural net architecture you've most likely heard of in the news is the ***convolutional neural network (CNN)***

Makes the explicit assumption that the input is an image

Constrains the structure of the network in a sensible way to make learning the weights scalable to high-resolution images

***Main insight: Translation invariance***

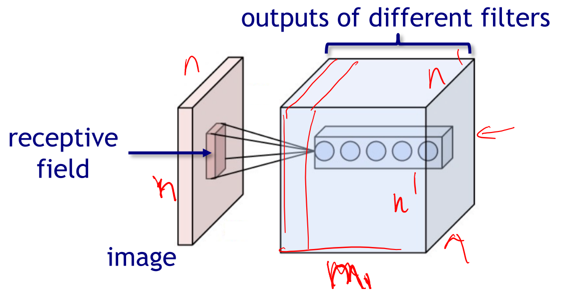
## Convolutional layer

The weights we apply to raw pixels should be the same, no matter where the object is located in the image

We begin by convolving the image with $M_1$ different filters

- filters are localized, the filter weights represent the parameters to be learned

The output of these filters are then passed through a ReLU

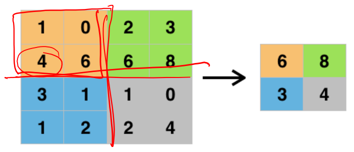


## Pooling layer

The other key concept in CNNs is ***pooling*** (downsampling)

Given the output of a filter, we can downsample the output to produce a smaller number of coefficients for the next layer

Most common choice is known as ***max pooling***

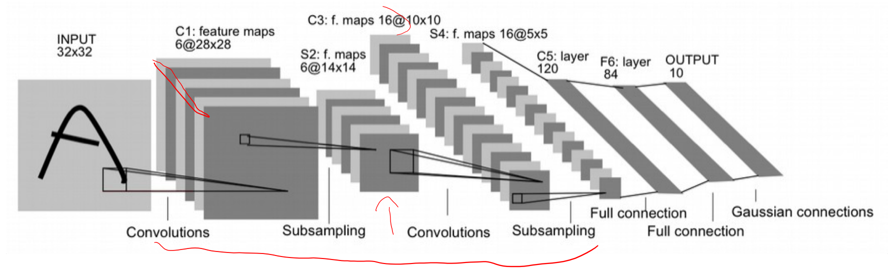


Intuition: the precise location of a feature is not important

A CNN will typically have a pooling layer after each convolution layer

## Full CNN

Yann LeCun (1998)



Note that as you go deeper and subsample more, there are more filters/feature maps available

After a certain depth, the network simply consists of fully connected layers

## Other deep architectures

- Recurrent neural networks
  - output of hidden nodes feeds back as input to other hidden nodes
  - used in time-series prediction
- Autoencoders
  - input and output are the same, but number of hidden nodes is constrained to be small relative to input dimension
  - form of dimensionality reduction
- Deep belief networks, restricted Boltzman machines, …
  - graphical models, probabilistic autoencoders
- …

## When to use deep learning?

***As a last resort!***

In all seriousness, I suggest tackling a new problem by starting with simpler models first

1. Try a linear classifier (logistic regression/linear SVM)
2. If that doesn’t work, try
   - nearest neighbors if you have lots of data in low-dimensions
   - naïve Bayes if you have data (esp text) in high-dimensions
3. If that still doesn’t work, try
   - random forests
   - support vector machines with a polynomial or rbf kernel
   - boosting
4. If you’ve convinced yourself that none of the above work well enough, then you are justified in going deep…