Deep Learning Theory and Applications

Gradient Descent

Yale

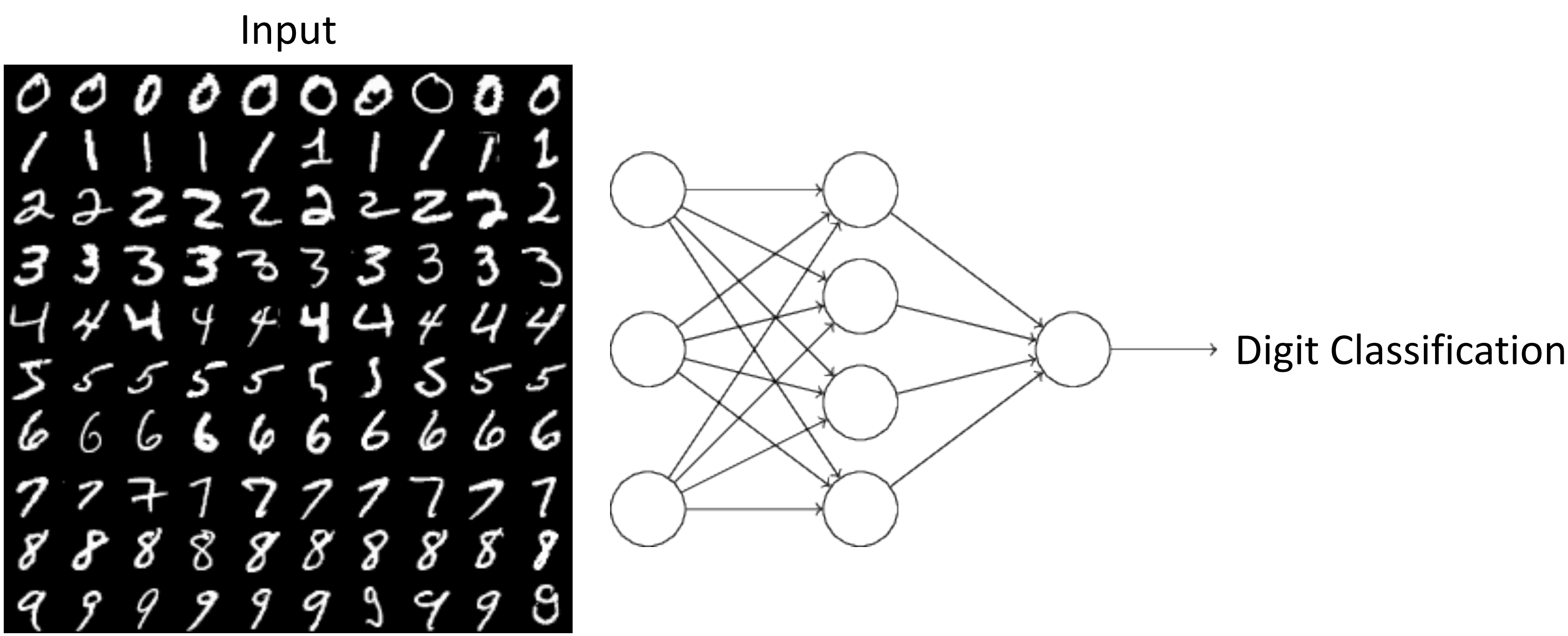
CPSC/AMTH 663

Learning with Gradient Descent

MNIST as an example

Classifying handwritten digits

- Goal: learn the weights and biases in a network to perform handwritten digit recognition



MNIST data set

- 60,000 training images
  - 28×28 greyscale images
  - Scanned handwriting samples from 250 people (half from US Census Bureau employees, other half were high school students)
- 10,000 test images
  - 28×28 greyscale images
  - Scanned handwriting samples from different 250 people (half from US Census Bureau employees, other half were high school students)
- $x =$ a training input
  - $28 \times 28 = 784$-dimensional vector
- $y = y(x)$ is the desired output
  - 10-dimensional vector
  - E.g., if x depicts a 6, then $y(x) = (0,0,0,0,0,0,1,0,0,0)^T$

Cost function

- Goal: develop an algorithm that finds weights and biases s.t. the network output approximates $y(x)$ for all training inputs x
- Define a ***cost function*** (also referred to as a loss or objective function)

$$C(w, b) = \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

  - w is the collection of all weights in the network
  - b is the collection of all biases in the network
  - $a = a(x, w, b)$ is the output when x is the input
  - C is referred to as the quadratic cost function or mean squared error (MSE)

Cost function

$$C(w,b) = \frac{1}{2n}\sum_x \|y(x) - a\|^2$$

- w is the collection of all weights in the network
- b is the collection of all biases in the network
- $a = a(x,w,b)$ is the output when x is the input

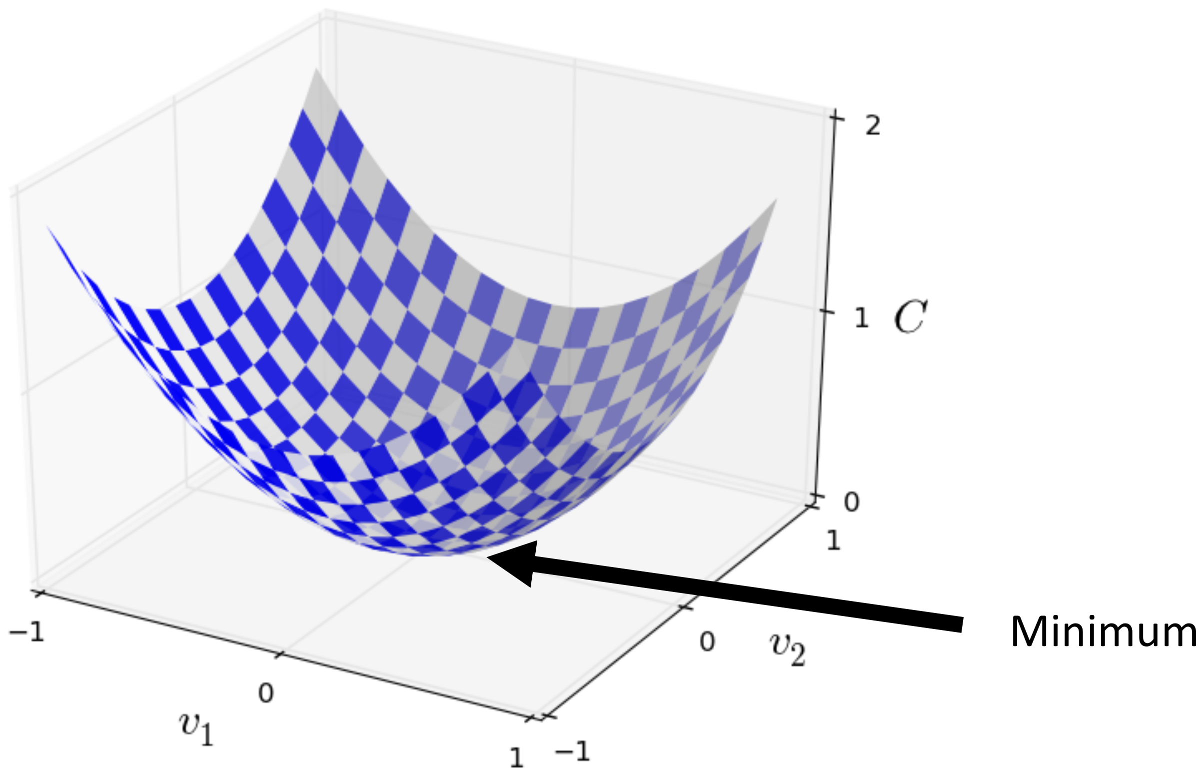
- $C(w,b) \approx 0$ when $y(x) \approx a$ for all training inputs x
  - Algorithm has done well if it can find weights and biases s.t. $C(w,b) \approx 0$
  - Algorithm has not done well if $C(w,b)$ is large
    - $\Rightarrow y(x)$ is not close to the output a for a large number of inputs
- Goal of training algorithm is to choose w and b to minimize $C(w,b)$
- We'll use ***gradient descent***

Cost function

- Why use MSE?
  - Why not use 0-1 loss or average probability of error?
- 0-1 loss is not a smooth function of the weights and biases
  - Difficult to change weights and biases to improve performance as measured by this loss
- Small changes in weights and biases can result in small improvements to MSE
  - We can figure out which direction to go

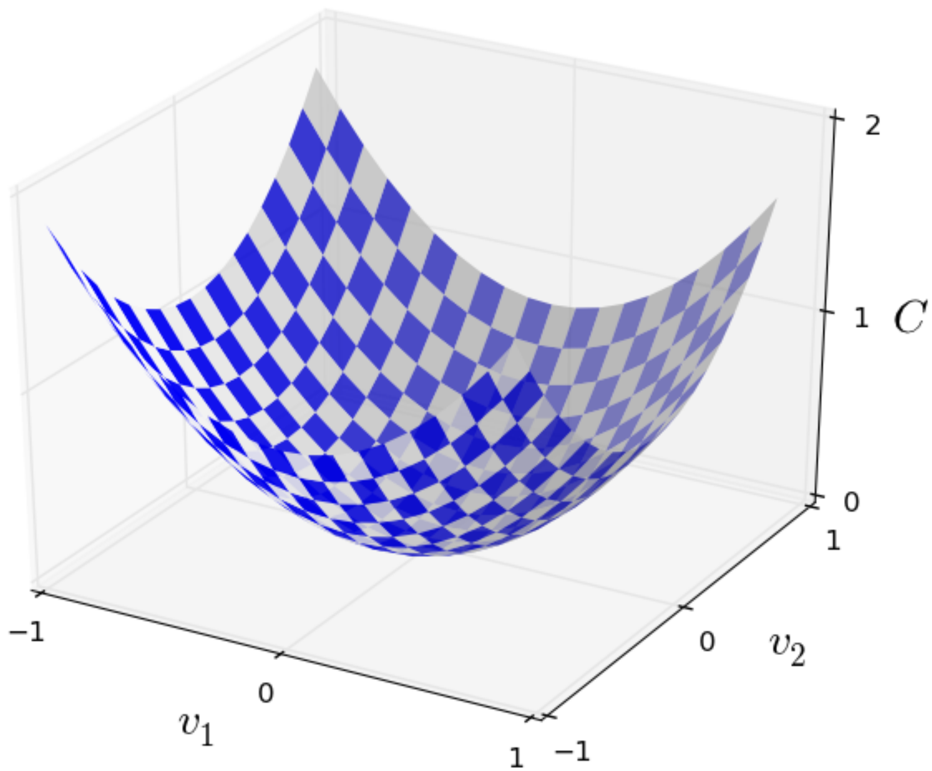
Gradient descent

- How do we minimize a cost function in general?
- Suppose we want to minimize some function $C(v)$
  - $v = v_1, v_2, \ldots$
  - We're not limiting ourselves to neural networks right now



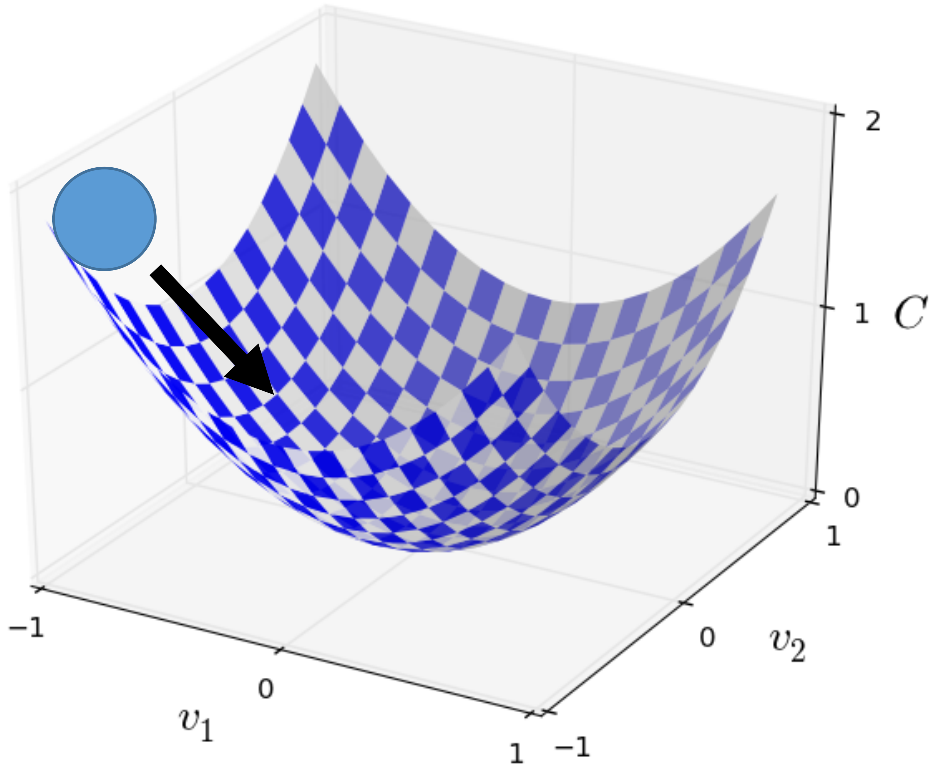
Gradient descent

- What if we have more variables?
- Could try calculus to find extremum of C
  - Difficult when we have lots of variables
  - Largest neural networks have billions of weights and biases



Rolling ball analogy

- Choose a random start point
- Ball rolls to the bottom of the valley
- Can simulate this by computing 1st and 2nd derivatives of C

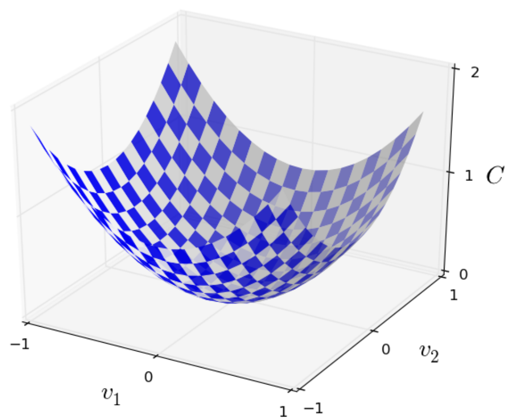


The gradient

- Choose a random starting point
- Move small amounts Δv_1 in the v_1 direction and Δv_2 in the v_2 direction

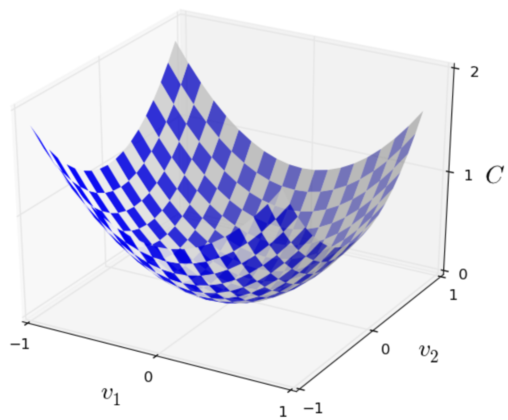
$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

- Goal is to minimize C
  - Choose Δv_1 and $\Delta \mathrm{v}_2$ so that ΔC is negative
  - I.e. move in a direction that decreases C
- Define $\Delta v = (\Delta v_1, \Delta v_2)^T$
- The ***gradient*** of C is $\nabla C = \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$
- $\Delta C \approx \nabla C \cdot \Delta v$



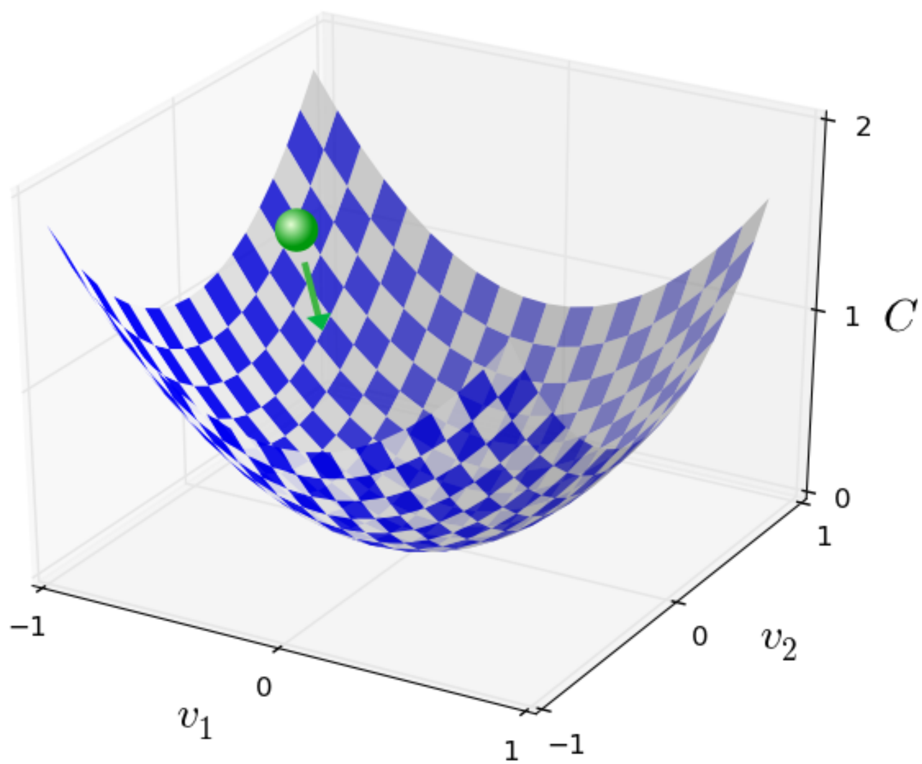
The gradient

- Define $\Delta v = (\Delta v_1, \Delta v_2)^T$
- The ***gradient*** of C is $\nabla C = \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$
- $\Delta C \approx \nabla C \cdot \Delta v$
- How should we choose Δv to ensure ΔC is negative?
- Choose $\Delta v = -\eta \nabla C$
  - η is a small, positive parameter (known as the ***learning rate***)
  - $\Rightarrow \Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$
  - Since $\|\nabla C\|^2 \geq 0$, $\Delta C \leq 0$
  - Thus C will always decrease and never increase
- Thus, choose new position $v \rightarrow v - \eta \nabla C$
- Repeat until we reach the minimum



Summary of gradient descent

- Compute the gradient ∇C
- Move in the *opposite* direction
  - It's like falling down the slope of the valley



Gradient descent with more variables

- Gradient descent extends easily to functions of more variables
- Let C be a function of m variables, $v_1, \ldots, v_m$

$$\Delta v = (\Delta v_1, \ldots, \Delta v_m)^T$$

$$\nabla C = \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_m}\right)^T$$

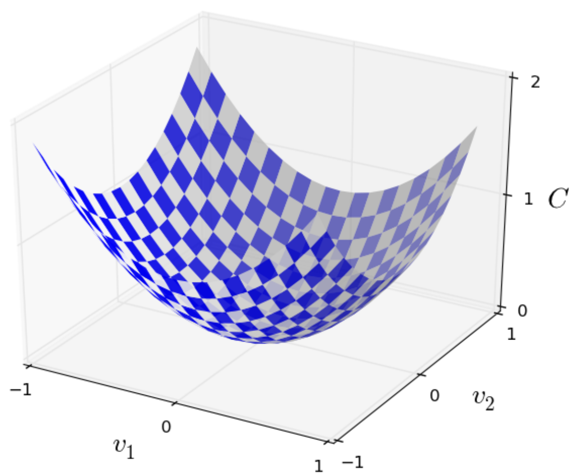
- Then $\Delta C \approx \nabla C \cdot \Delta v$
- As before, choose $\Delta v = -\eta \nabla C$
- The update step is again $v \rightarrow v - \eta \nabla C$

Gradient descent is optimal in some sense

- Suppose we want to find Δv that decreases C as much as possible
  - Equivalent to minimizing $\Delta C \approx \nabla C \cdot \Delta v$
- Constrain the size $\|\Delta v\| = \epsilon$ for some $\epsilon > 0$
- What is the direction that decreases C the most?
- Can be shown that $\Delta v = -\eta \nabla C$ minimizes $\nabla C \cdot \Delta v$
  - $\eta = \frac{\epsilon}{\|\nabla C\|}$
- Gradient descent can be interpreted as taking small steps in the direction that decreases C the most

Choosing the learning rate

- Need to choose η small enough that the approximation $\Delta C \approx -\eta \nabla C \cdot \nabla C$ is good
  - Otherwise, we may end up with $\Delta C > 0$
- On the other hand, very small $\eta \Rightarrow$ tiny steps
  - Slow convergence to the minimum
- In practice, η is varied so that the approximation $\Delta C \approx -\eta \nabla C \cdot \nabla C$ is good and the algorithm isn't too slow



Gradient descent in neural networks

- Goal: use gradient descent to find weights and biases that minimize $C(w,b) = \frac{1}{2n}\sum_x \|y(x) - a\|^2$
- Gradient descent update rules:

$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}$$

- Repeatedly applying the update enables us to “roll down the hill” to the minimum of the cost function

Gradient of a sigmoidal neuron

- $\mathrm{f}(z) = \frac{1}{1+e^{-z}}$
- Use chain rule with g(z) = e^{-z}
- $\frac{\delta(f(g(z)))}{\delta(z)} = \frac{\delta(f(g(z)))}{\delta(g(z))} \frac{\delta(g(z))}{\delta(z)}$
- $\frac{\delta(f(g(z)))}{\delta(z)} = \frac{e^{-z}}{(1+e^{-z})^2}$
- $\frac{\delta(f(g(z)))}{\delta(z)} = \frac{1+e^{-z}-1}{(1+e^{-z})^2}$
- $\frac{\delta(f(g(z)))}{\delta(z)} = \frac{(1+e^{-z})}{(1+e^{-z})^2} - \frac{1}{(1+e^{-z})^2}$
- $\frac{\delta(f(g(z)))}{\delta(z)} = \frac{1}{1+e^{-z}} - \frac{1}{(1+e^{-z})^2}$

- Nice form of answer is $\frac{\delta(f(g(z)))}{\delta(z)}$=f(z) (1-f(z))
- How do you get this in terms of weights and biases?

Large Sample Challenges

- Challenge: $C(w,b) = \frac{1}{n}\sum_x C_x$ is an average over training example costs $C_x = \frac{\|y(x)-a\|}{2}$
- $\Rightarrow \nabla C = \frac{1}{n}\sum_x \nabla C_x$
  - Computing the gradients for each training input x can be slow for large sample sizes
  - $\Rightarrow$ learning occurs slowly
- We can speed things up by computing ∇C_x for a small random sample of training inputs
  - Provides an estimate of ∇C
  - Speeds up gradient descent and learning
  - Known as ***Stochastic Gradient Descent*** (SGD)

Stochastic Gradient Descent (SGD)

- Pick a random set of m training inputs $X_1, X_2, \ldots, X_m$
  - Referred to as a ***mini-batch***
- If m is large enough, then the average value of ∇C_{X_j} will be roughly equal to the average over all ∇C_x:

$$\frac{1}{m}\sum_{j=1}^{m} \nabla C_{X_j} \approx \frac{1}{n}\sum_{x} \nabla C_x = \nabla C$$

- In neural networks, this gives update steps of

$$w_k \rightarrow w_k' = w_k - \frac{\eta}{m}\sum_{j} \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l \rightarrow b_l' = b_l - \frac{\eta}{m}\sum_{j} \frac{\partial C_{X_j}}{\partial b_l}$$

Stochastic Gradient Descent Summary

A training ***epoch***:

1. Pick a random subset of the training data
   - Referred to as a mini-batch
2. Update the weights and biases using the gradient estimates from the mini-batch
3. Pick another random mini-batch from remaining training points and repeat step 2
   - Repeat until all training inputs have been used

Repeat multiple epochs until stopping conditions

Analogy to political polling

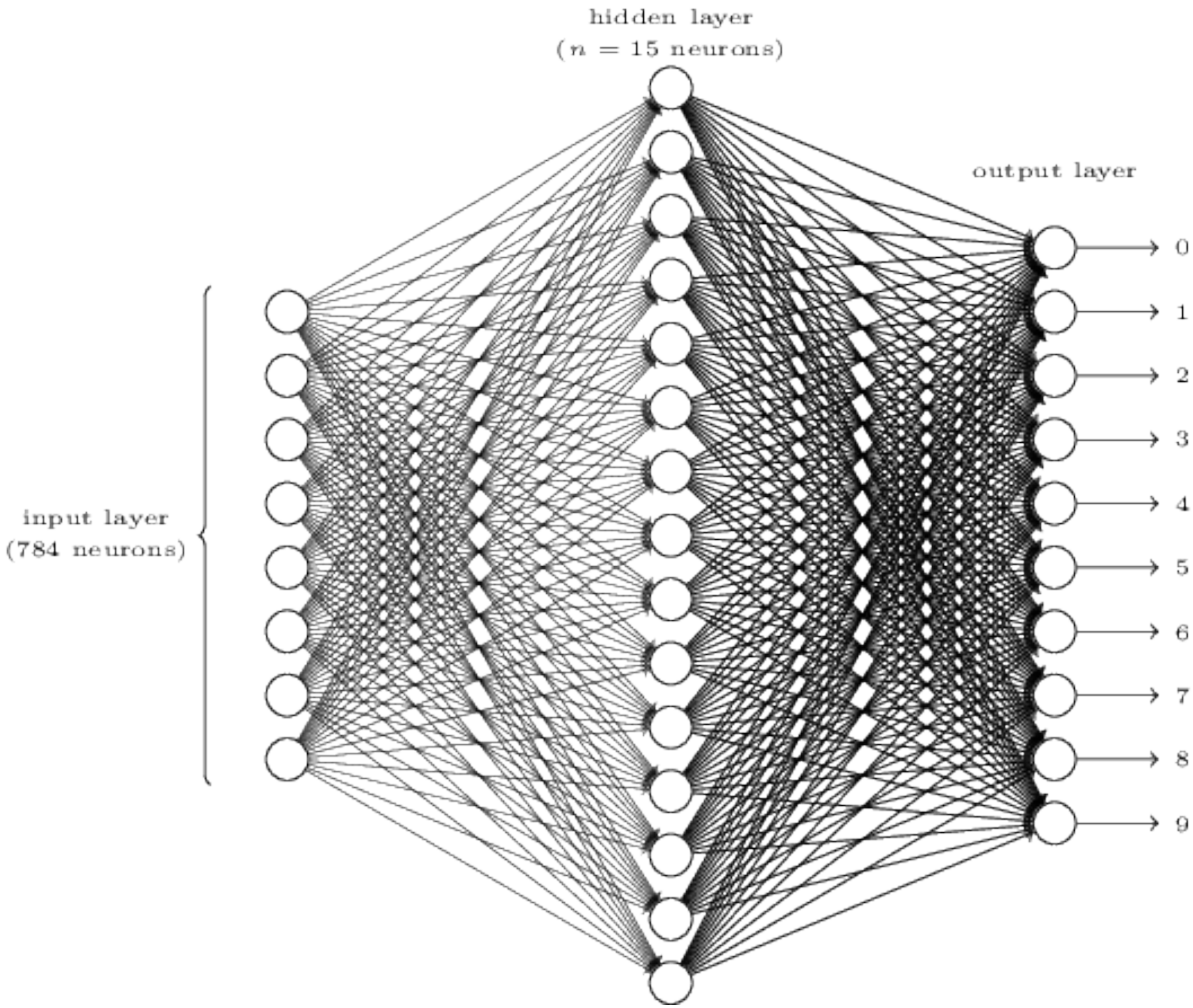
- Much easier to carry out a poll than run a full election
- Similarly, it's much easier to estimate gradients from mini-batches than the entire training set
- Downside: gradient estimates will be noisier in SGD
- That's ok: we only need to move in a general direction that decreases C
  - Don't need an extremely accurate estimate of the gradient
- In practice, SGD is used extensively in learning neural networks

Back to MNIST

- Goal: learn the weights and biases in a network to perform handwritten digit recognition
- Code and data available on github
  - Link provided in the Nielsen book, chapter 1
  - An optional (ungraded) problem in HW 1 is to implement this code on your machine
- First split the *training* data into 2 parts
  - 50,000 images for training
    - Referred to as the "MNIST training data"
  - 10,000 images for validation
    - Used to set ***hyper-parameters***
    - Not used today, but later

Our architecture

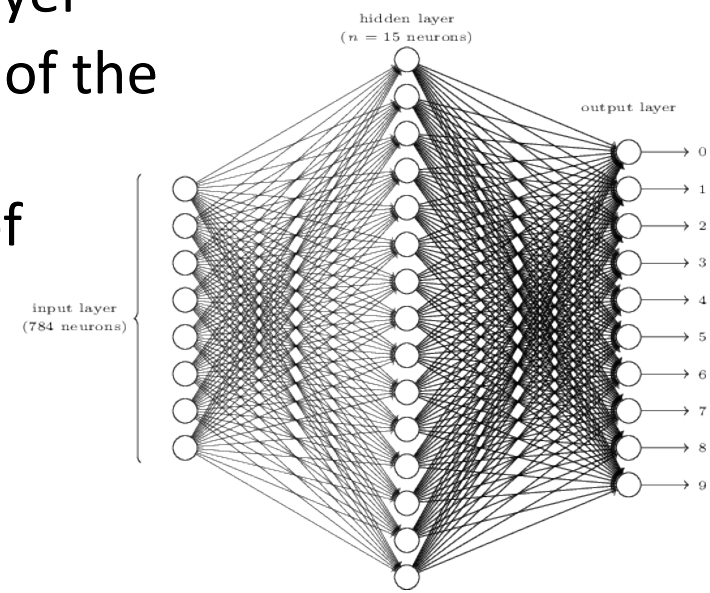
- We will vary the number of neurons in the hidden layer



Vector notation

- Store weights and biases as matrices
- Suppose we are considering the weights from the second (hidden) layer to the third (output) layer
  - w is the weight matrix with w_{jk} the weight for the connection between the kth neuron in the second layer and the jth neuron in the third layer
  - b is the vector of biases in the third layer
  - a is the vector of activations (output) of the 2nd layer
  - a' the vector of activations (output) of the third layer

$$a' = \sigma(wa + b)$$



How the code works

1. In each epoch, randomly shuffle the training data
2. Partition the shuffled training data into mini-batches
3. For each mini-batch, apply a single step of gradient descent
   - Gradients are calculated via ***backpropagation*** (the next topic)
4. Train for multiple epochs

First attempt

- 30 nodes in hidden layer
- 30 epochs
- Mini-batch size $= 10$
- Learning rate $\eta = 3.0$

```
Epoch 0: 9129 / 10000
Epoch 1: 9295 / 10000
Epoch 2: 9348 / 10000
...
Epoch 27: 9528 / 10000
Epoch 28: 9542 / 10000
Epoch 29: 9534 / 10000
```

Second attempt

- **100 nodes in hidden layer**
- 30 epochs
- Mini-batch size $= 10$
- Learning rate $\eta = 3.0$
- Improves accuracy to 96.59%
  - Although depends on initialization: some runs give worse results

Third attempt

- 100 nodes in hidden layer
- 30 epochs
- Mini-batch size $= 10$
- **Learning rate $\boldsymbol{\eta = 0.001}$**

```
Epoch 0: 1139 / 10000
Epoch 1: 1136 / 10000
Epoch 2: 1135 / 10000
...
Epoch 27: 2101 / 10000
Epoch 28: 2123 / 10000
Epoch 29: 2142 / 10000
```

Debugging a neural network

- What do we do if the output is essentially noise?
- Suppose we ran the following as our first attempt:
- 30 nodes in hidden layer
- 30 epochs
- Mini-batch size $= 10$
- Learning rate $\eta = 100.0$

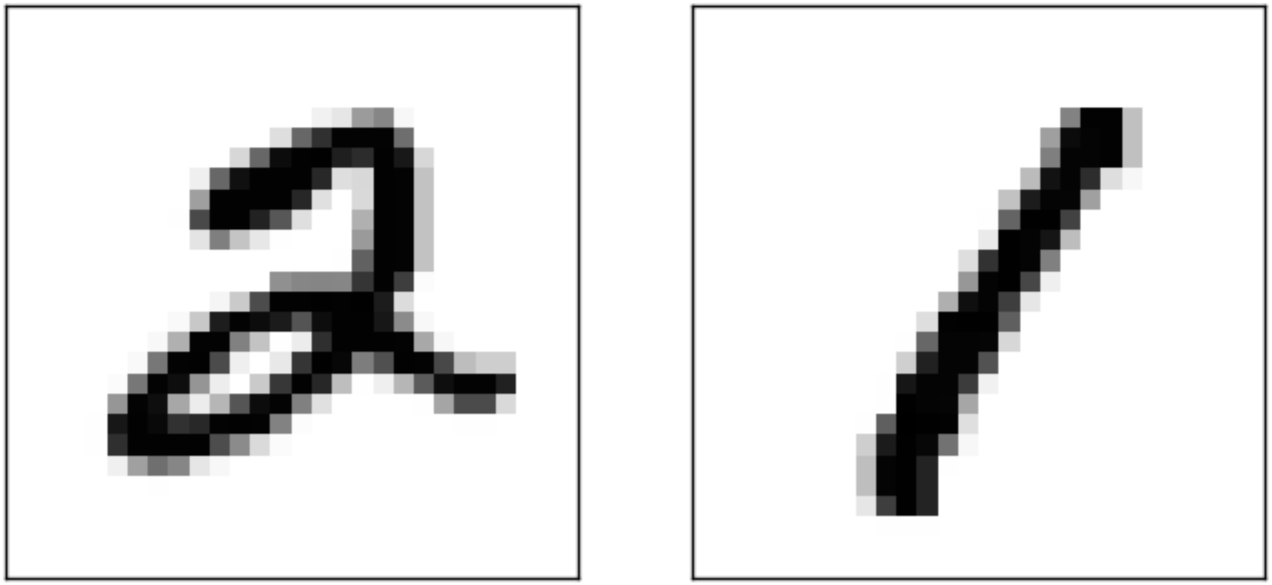
```
Epoch 0: 1009 / 10000
Epoch 1: 1009 / 10000
Epoch 2: 1009 / 10000
Epoch 3: 1009 / 10000
...
Epoch 27: 982 / 10000
Epoch 28: 982 / 10000
Epoch 29: 982 / 10000
```

Debugging a neural network

- What can we do?
  - Should we change the learning rate?
  - Should we initialize differently?
  - Do we need more training data?
  - Should we change the architecture?
  - Should we run for more epochs?
  - Are the features relevant for the problem (i.e. is the Bayes error rate reasonable)?
- Debugging is an art
  - We’ll develop good heuristics for choosing good architectures and hyper parameters

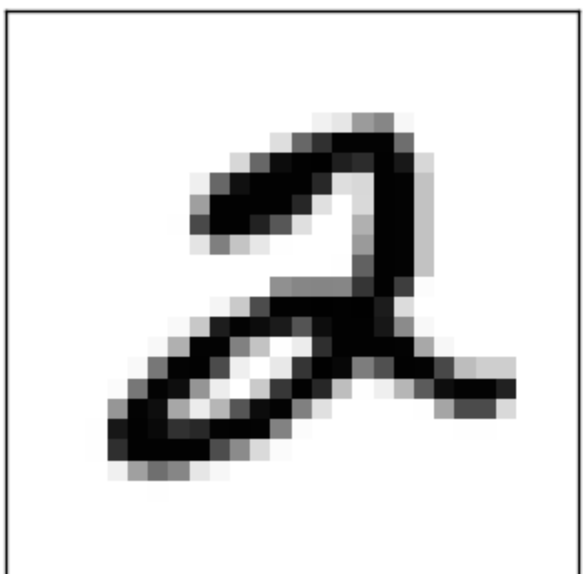
How well does our network do?

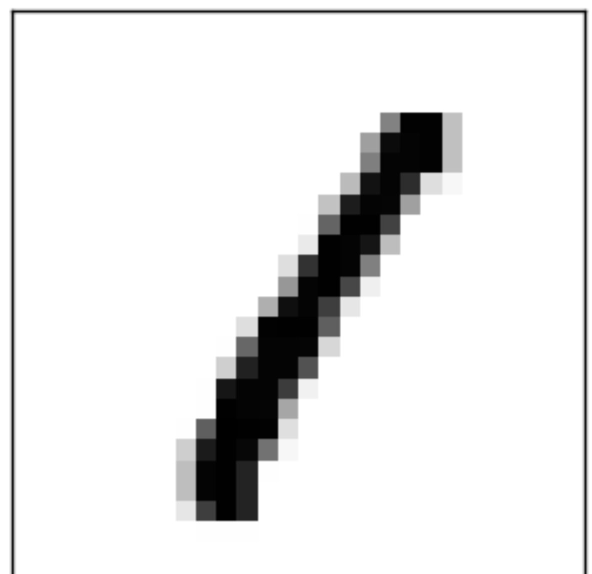
- Need to compare to some baselines
- Random guessing: 10% accuracy
  - Our network does much better
- Simple idea: How dark is the image?
  - E.g. a 2 will typically be darker than a 1



Average darkness

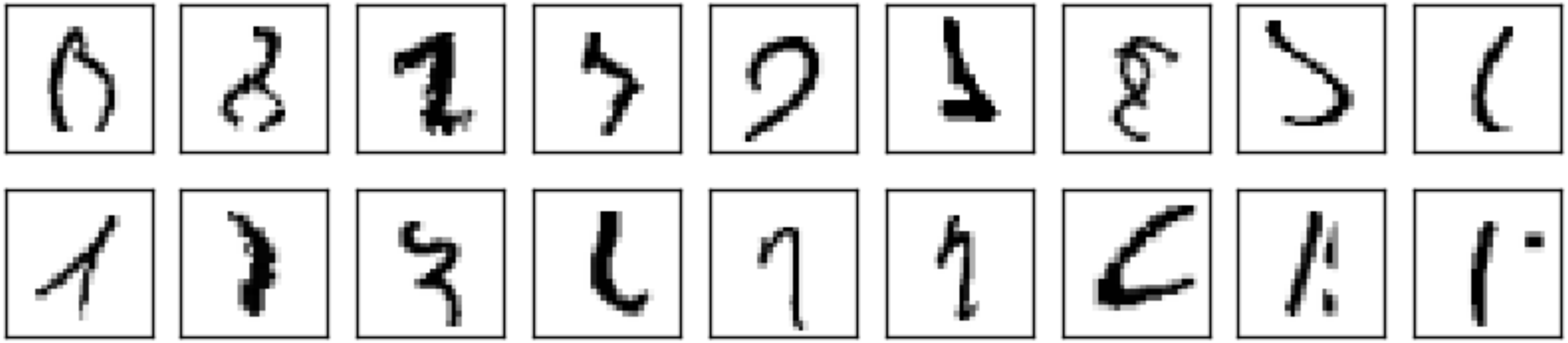
- Compute average darkness for each digit from the training data
- Classify an image based on the digit with the closest average darkness
- This gives 22.25% accuracy
  - Better than random guessing





Support Vector Machine (SVM)

- Using default settings in SVM scikit-learn (a Python library) gives 94.35% accuracy
  - It's possible to optimize tuning parameters to achieve 98.5% accuracy
- Can neural networks do better?
- Yes!
  - Record set in 2013 was 99.79% accuracy (only 21 wrong in the test data)
  - Can you do better?



Take away message

- We achieved pretty good results using simple architecture and off-the-shelf SVM
- State of the art network was also pretty simple (compared to modern architectures)

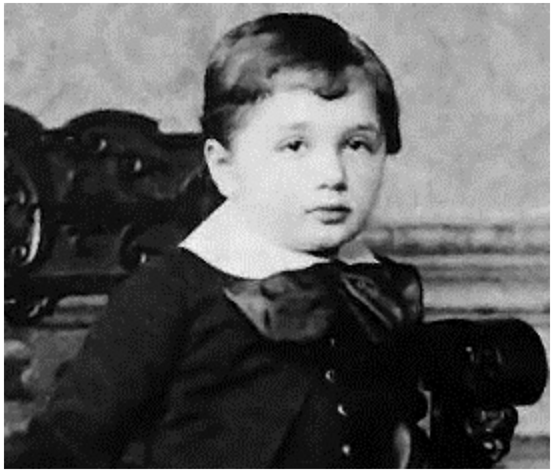
$$\text{sophisticated algorithm} \leq \text{simple learning algorithm} + \text{good training data}$$

Why does the neural network work?

- Weights and biases were discovered automatically
- Can we find a way to understand how the network classifies the digits correctly?
  - Given this understanding, can we improve?
- More broadly, suppose we develop artificial intelligence in the future
- Will we understand how such intelligence works?
- Will such an understanding (if we obtain it) lead to better understanding of the brain?
  - This was a goal of early AI research

Some heuristics

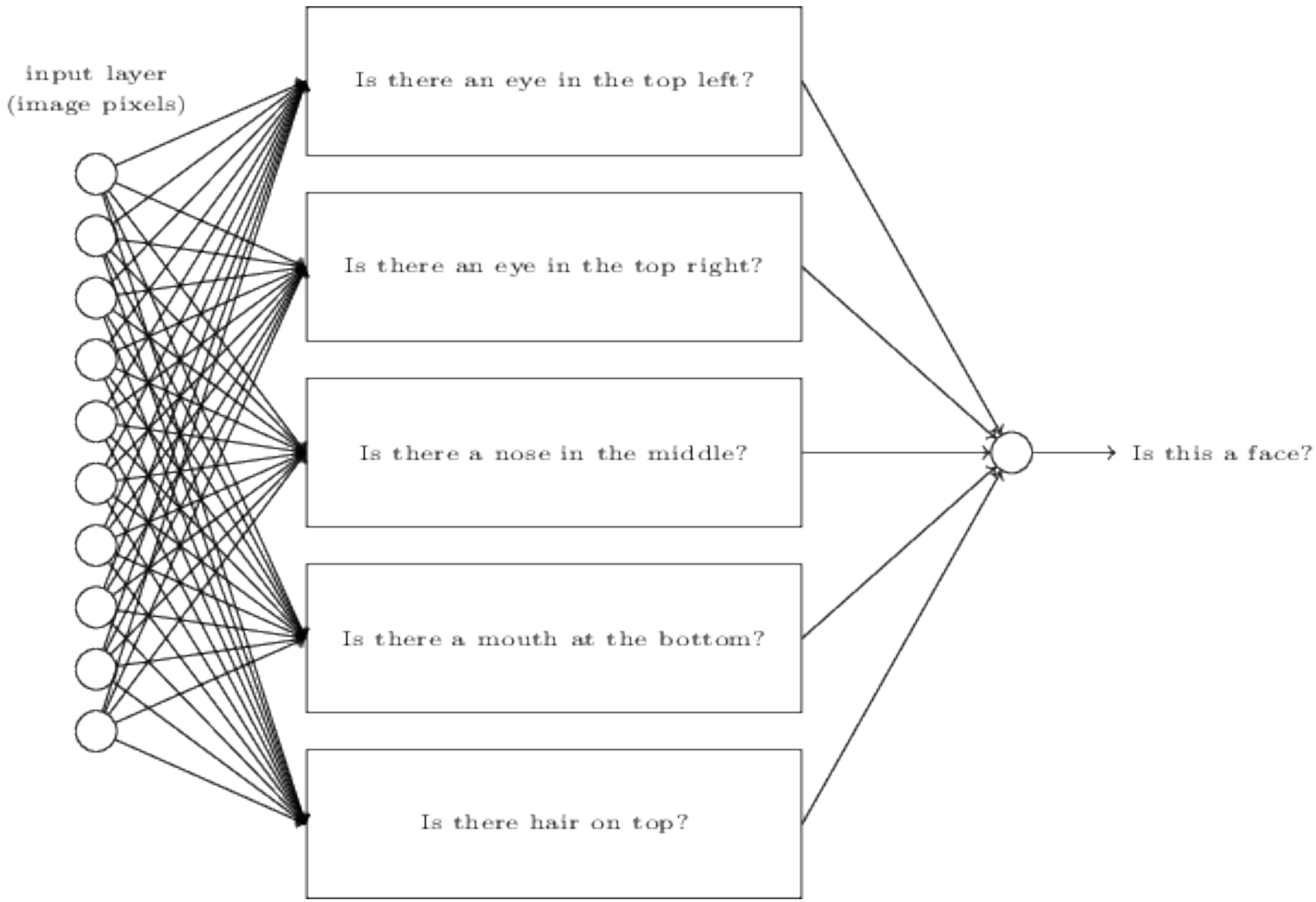
- Suppose we want to detect a human face in images


- How would we design a network by hand?

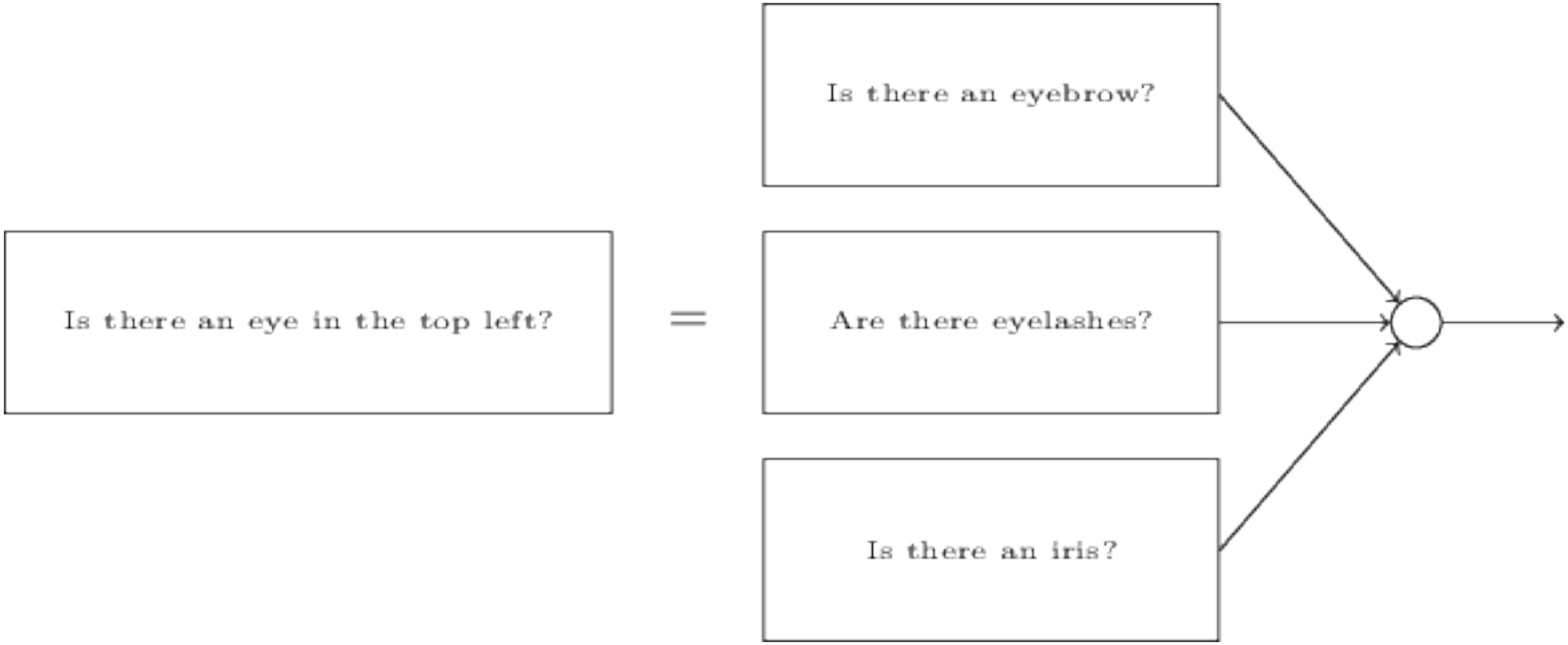
Some heuristics

- Break the problem down into sub-problems



Some heuristics

- Can break the sub-problems down further



- Eventually, we deal with sub-problems at the pixel level

Towards Deep learning

- The end result is a network that answers a complicated question
  - Uses a series of many layers
  - Early layers answer simple questions about input image
  - Later layers build up a hierarchy of complex and abstract concepts
  - Known as a ***deep neural network***
- Researchers tried to train deep neural networks in the 80’s and 90’s
  - Little luck
- Breakthroughs since 2006 have made it much easier to train deep networks

Further Reading

- Nielsen, chapter 1
- Goodfellow et al., Section 5.9 and chapter 6