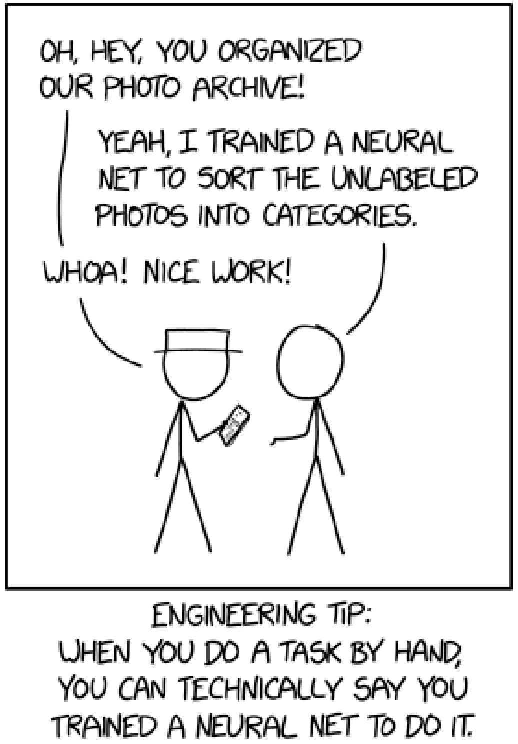
CAB420: Neural Networks and their Components

A BIT LIKE LEGO

Neural Networks

- Neural networks are a collection of layers
- In a simple network, layers connect to each other sequentially
  - Data flows from one layer to the next
- Overall structure inspired by the human brain
  - Activations cascading through the brain is mimicked by data propagating through the neural network



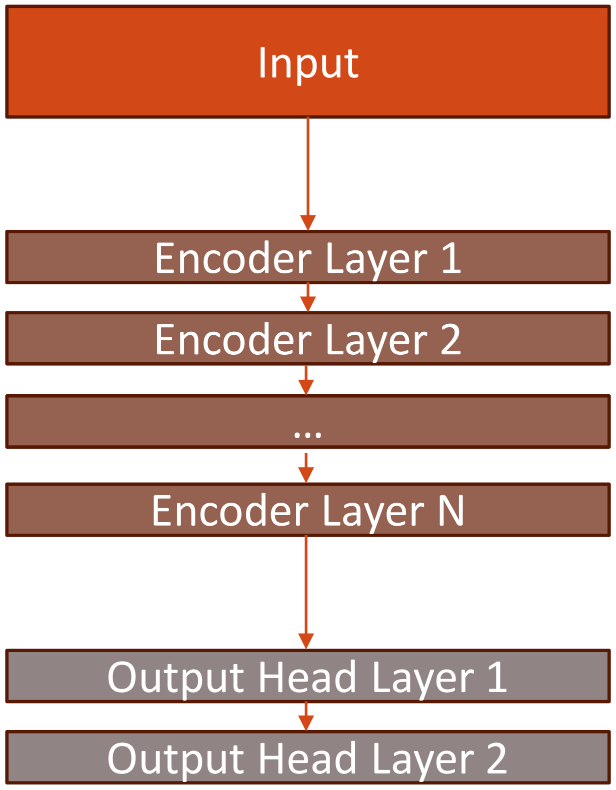
Cartoon from XKCD

High Level Components

THE DUPLO BLOCK WAY OF LOOKING AT THINGS

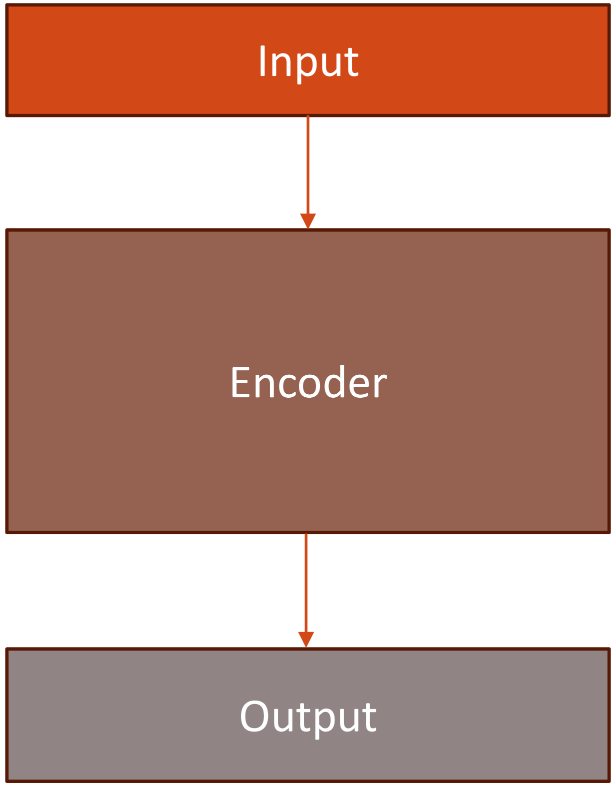
High-Level Network Structure

- Input
  - The input to the network
  - Typically, a fixed size
  - Can be images, audio, text, tables
    - If you can represent it as a number, it can be used
- Encoder/Backbone
  - Set of layers that encoders the input into some other representation
  - Can take many forms
- Output head
  - Take the encoded representation, estimate something
    - Regression, classification, anything you can craft an objective function for
  - Typically, a few layers at most
  - Has a loss function attached to learn the task



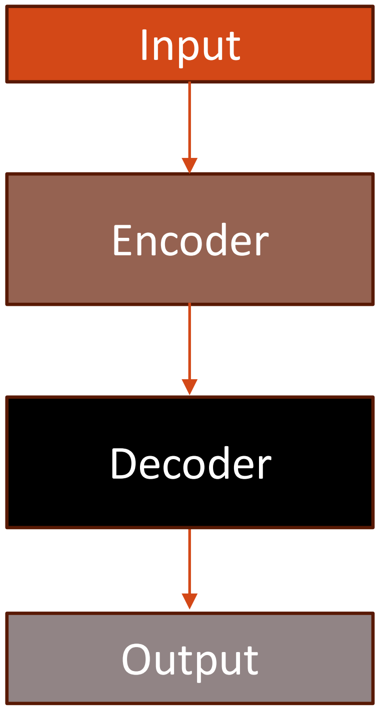
High-Level Network Structure

- We will often visualise and think of networks in terms of these high-level components



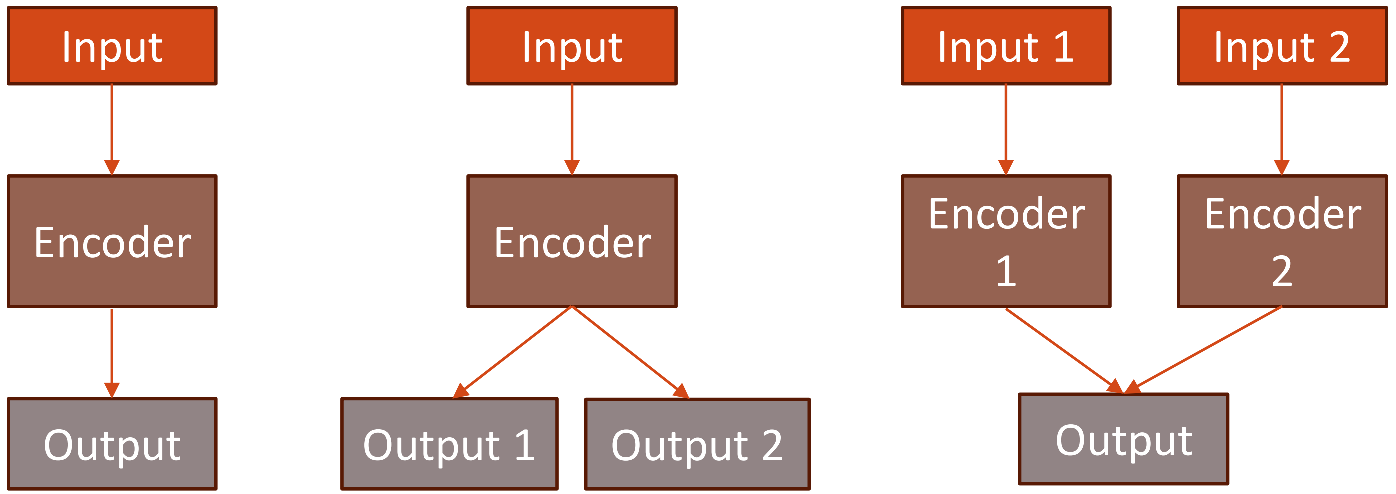
High-Level Network Structure

- The other type of component we'll commonly encounter is the Decoder
  - Takes some encoder representation, decodes it to something else
  - Commonly used when mapping from one domain to another
    - Language translation
    - Neural Style Transfer (i.e. making your photos look like paintings)



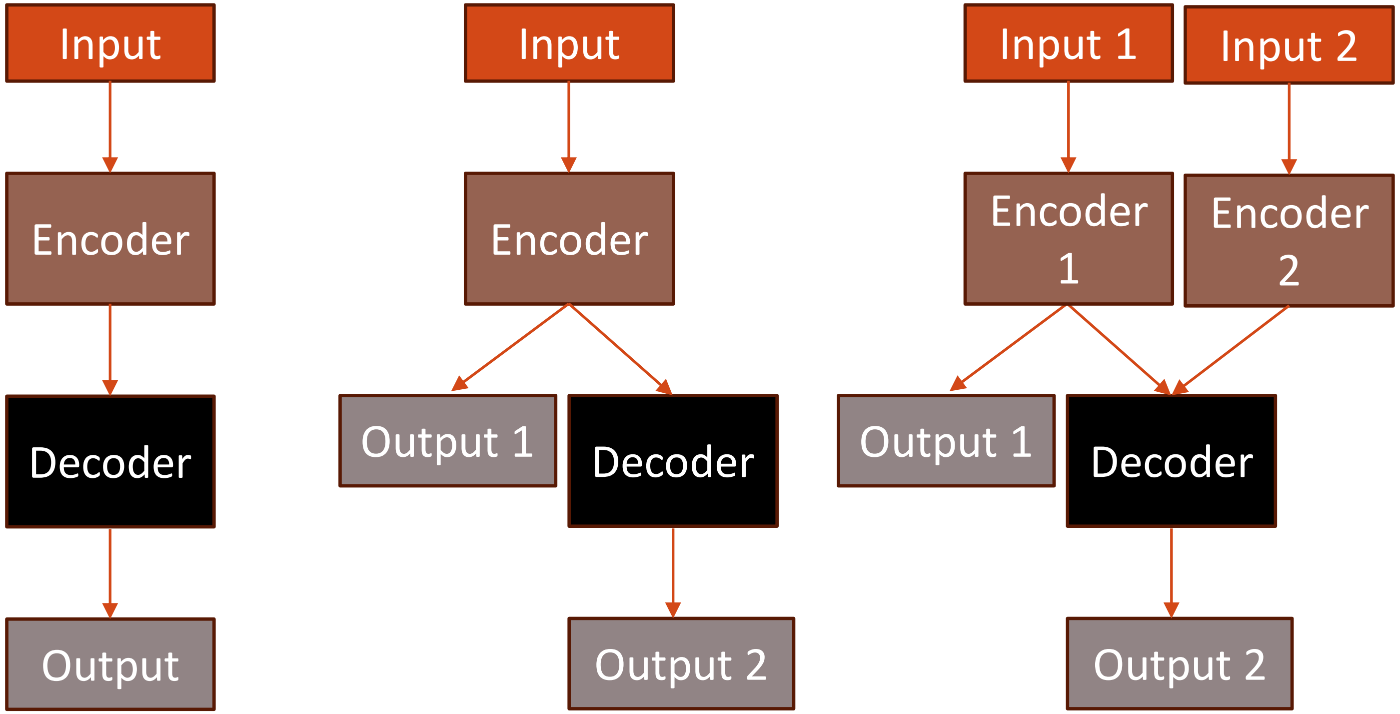
High-Level Network Structure

- We can arrange components in different ways to achieve different things



High-Level Network Structure

- We can arrange components in different ways to achieve different things



High Level Network Components

- Inputs and Outputs will typically be at least somewhat task specific
- Encoders and Decoders (or backbones) however may be quite general
- Practically this means we can take a network designed for one problem, and adapt it to a new problem by changing the input and output
  - The encoder (and decoder if there is one) stays the same

Encoder (and Decoder) Types

- Our encoders (and decoders) are a stack of layers
- Using different layers, in different ways, gives us different network types
  - Also referred to as Architectures
- Within Deep Learning, we have three common families of network types
  - Fully connected networks
    - All dense layers
    - Early approach, rarely used now
  - Convolutional neural networks (CNNs)
    - Uses convolutional layers
    - Largely responsible for the deep learning and AI boom
    - Still prominent, though no longer state of the art
  - Transformers
    - Current state of the art architecture

Encoder (and Decoder) Types

- In CAB420 we'll mostly use CNNs
- CNNs offer us:
  - Better performance than fully connected networks
  - Less computationally demanding than Transformers
  - Continue to be extremely prominent in real-world applications
- We'll look at transformers briefly towards the end of semester

Network Layers

LEGO RATHER THAN DUPLO

Neural Network Components

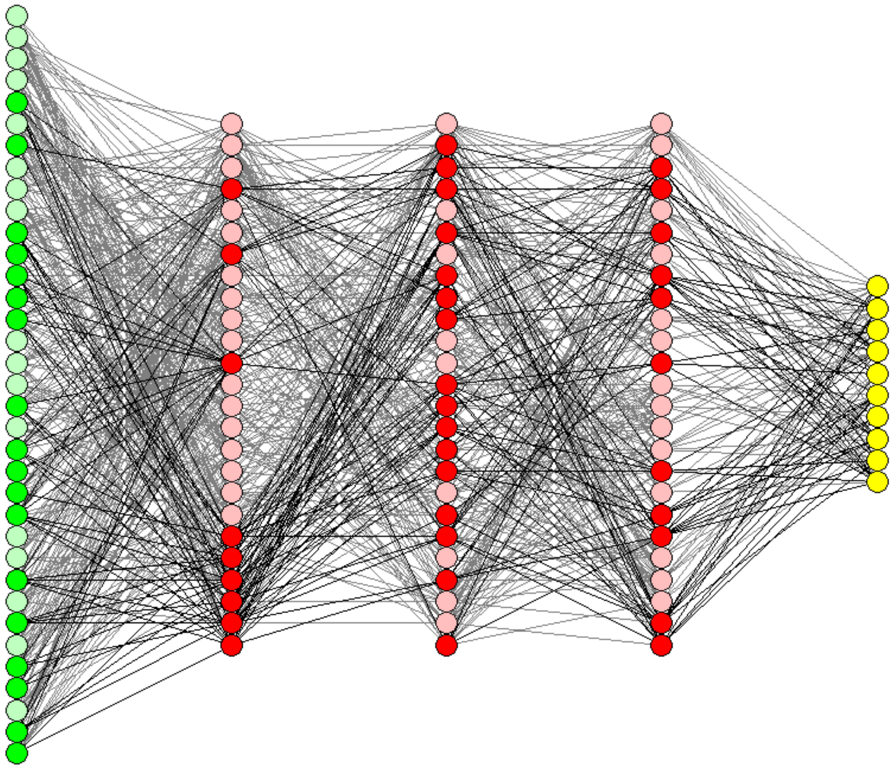
- Networks are built from a collection of layers
  - Layers are separated by non-linearities (a non-linear function)

- There lots of layers, but the main layers we'll consider are
  - Fully Connected Layer
  - Convolutional Layer
  - Pooling
  - Activation

- These layers give us the main building blocks for
  - Fully connected networks
  - CNNs
  - Transformers

Fully Connected Layers

- Every neuron in one layer is connected to every neuron in the next
  - Doesn't really capture spatial relationships in the data
    - i.e. Spatially adjacent neurons do not necessarily give similar results
    - Essentially a matrix multiplication



Fully Connected Layers

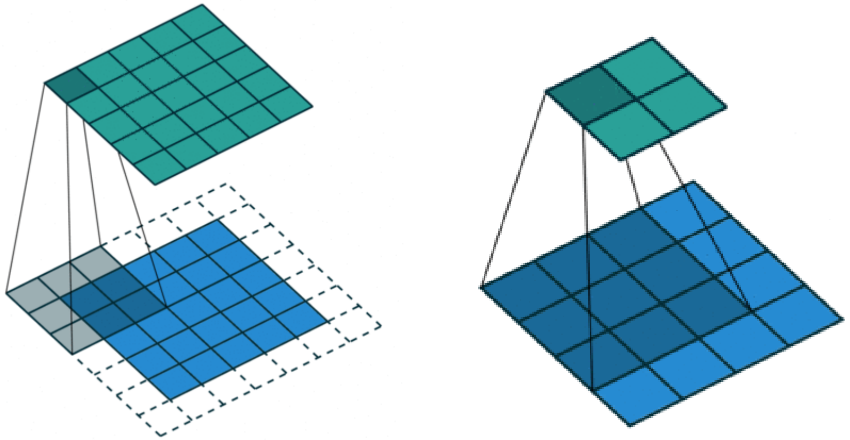
- Input: a vector of size [1 x M]
- Output: a vector of length [1 x N]
- Parameters:
  - Weight matrix, [M x N] in size
  - Bias vector, [1 x N] in size
- Computation:
  - Output = input*W + B
  - For large M and/or N, W becomes very large

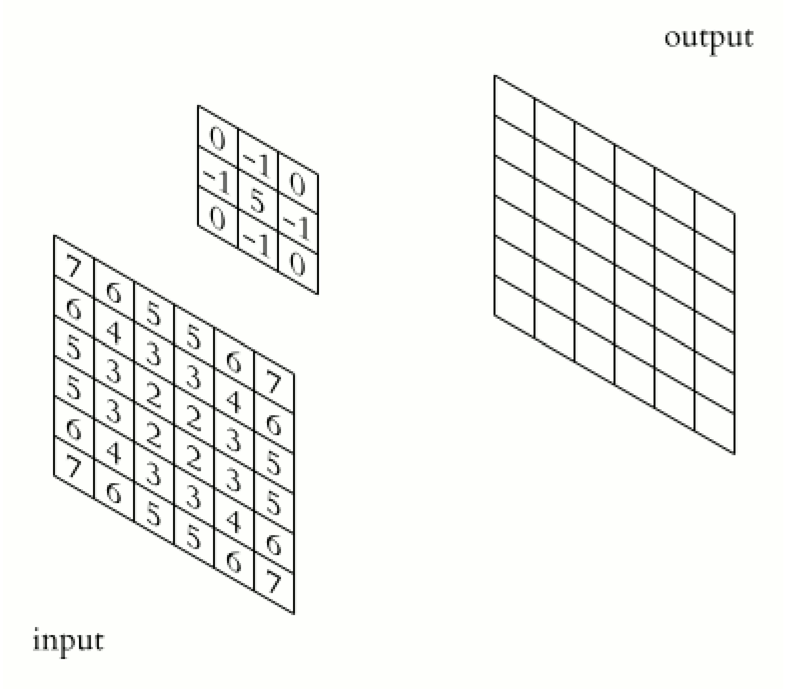
Convolutional Layer

- Convolution

$$(f * g)(t) \triangleq \int_{-\infty}^{+\infty} f(\tau)g(\tau - r)dr$$

  - The integral of the product of the two functions after one is reversed and shifted
  - Performs a weighted sum of one input according to the second
  - Typically viewed as a filtering process





Convolution

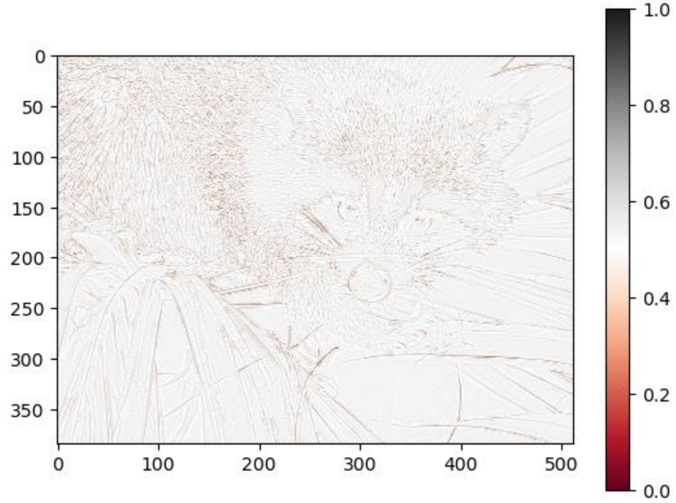
- With images, can be thought of as applying a filter

Input

Convolution Kernel

[-1 –1 –1
-1 8 -1
-1 –1 –1]

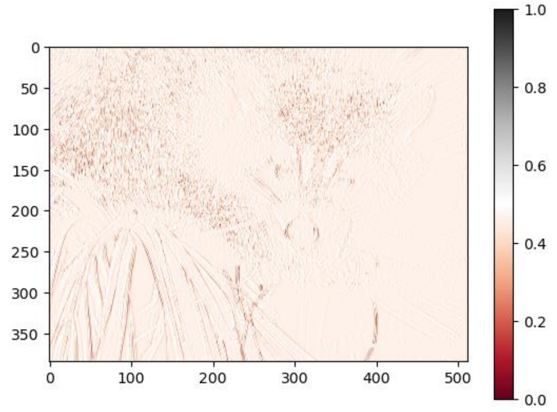
Output



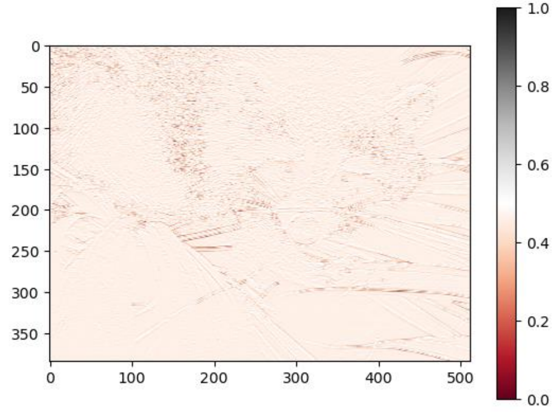
- Example kernel is an edge detection kernel
  - Detects all edges by finding pixels where there is a local change in contrast

Convolution

- Different Kernels will give different outputs
  - Vertical Edges
    - [-4 8 –4
      -4 8 –4
      -4 8 –4]

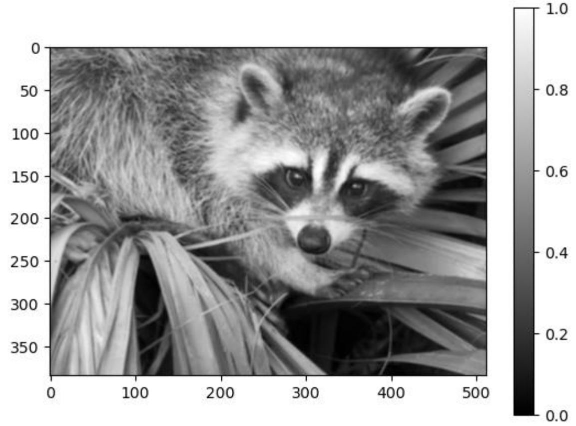


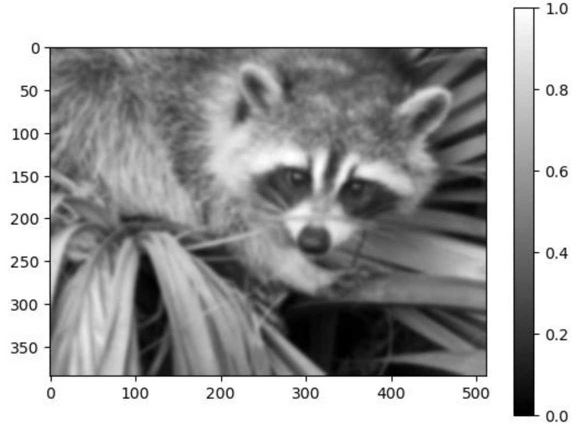
  - Horizontal Edges
    - [-4 –4 –4
      8 8 8
      -4 –4 –4]



Convolution

- The output of one convolution operation can be used as the input to another
  - Stacking filters
  - Blurring kernel:
    - [0.111 0.111 0.111
      0.111 0.111 0.111
      0.111 0.111 0.111]
  - Top image: after 1 blur
  - Bottom image: after 7 blurs

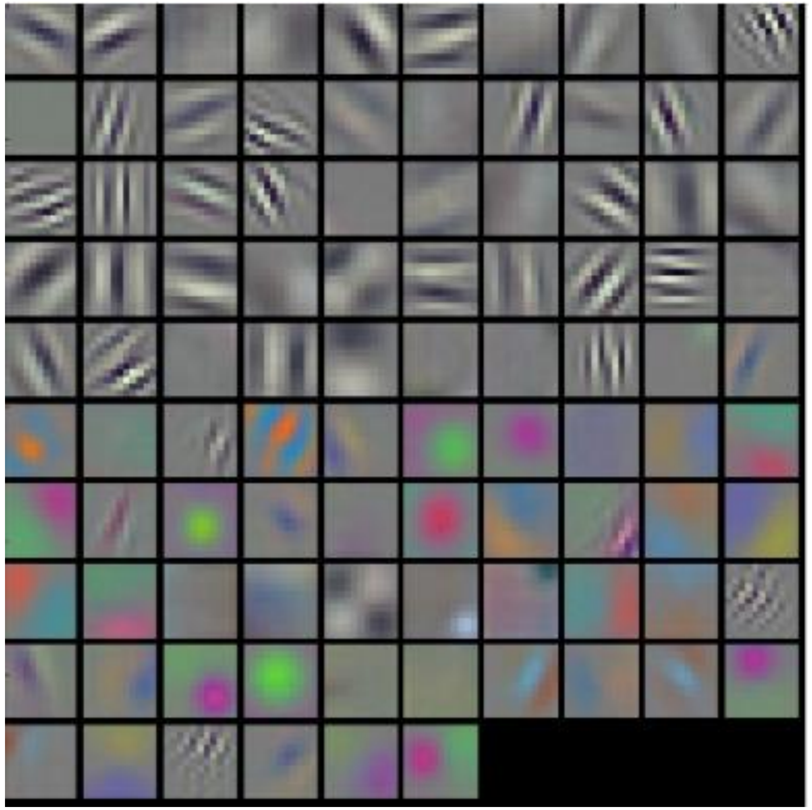




Example output taken from ***CAB420_DCNNs_Additional_Example_2_Convolutions.ipynb***

Convolutional Layer

- You don’t need to memorise a bunch of kernels!
- In neural networks
  - We learn the filters
  - Typically learn lots of filters at once
- Learned filters can
  - Represent simple shapes, edges, textures in early layers
  - Represent more complex structures in later layers
- Filters operate over all channels in the input
  - If we have a colour input, we have a colour filter

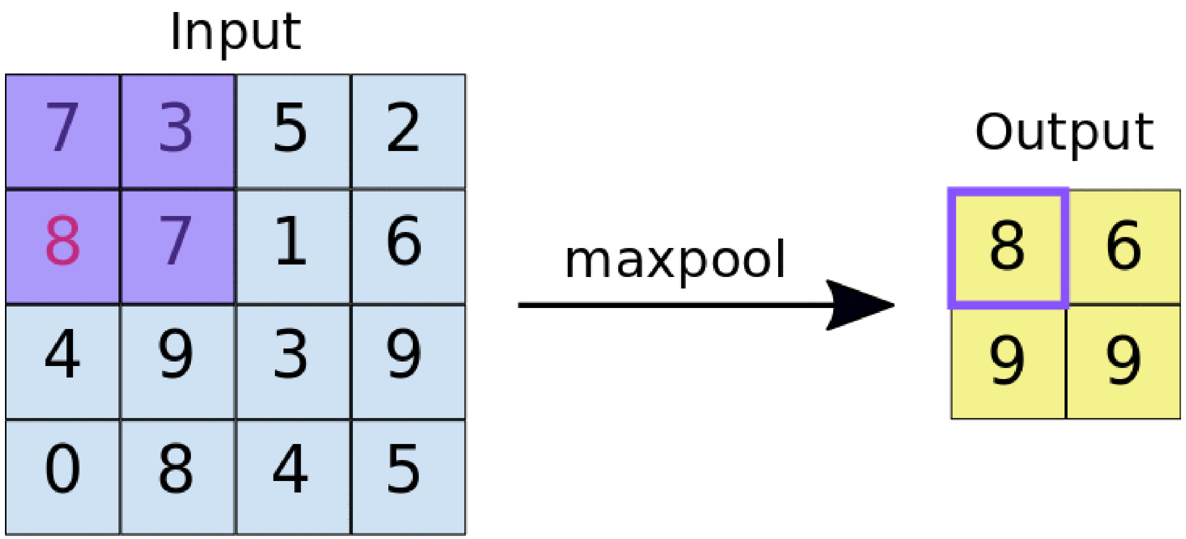


Convolution Layer

- Input: [W x H x C] image
- Output: [W x H x N] image
  - N is number of learned filters
- Parameters:
  - Each filter is [X x Y] in size, and has [X x Y x C] weights
  - 1 bias value per filter
- Computation:
  - Each filter applied at each location in target image, bias is added per filter
  - Operates over all image channels at once
  - Each filter results in one output channel in the output
- Other configuration options:
  - Stride: do we apply the filter at every pixel, or skip some?
  - Behavior at borders: do we pad such that all pixels can be used?

Pooling

- Used to aggregate features
  - Reduces dimensionality
- Multiple types of pooling
  - Min, Max, Average
  - We almost always use Max Pooling
    - Take the maximum value in a region as output
  - Typically placed after a convolution layer



Pooling

- Input: [W x H x C] image
- Output: [W' x H' x C] image
  - Output is reduced spatially, but number of channels is unchanged
- Parameters:
  - No learned parameters
  - Size and type of pooling operation is fixed when network is created

Activations

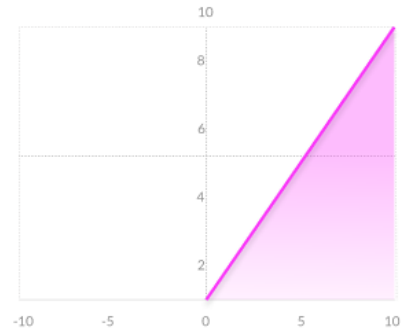
- Neural networks connect outputs of one layer to the inputs of the next
- However, we don't just feed them straight in, we pass them through an "activation function"
  - Can be seen to turn some neurons on, and others off
  - Introduces non-linearities, helpful for learning complex functions
  - Different activations let different amounts of information flow
    - Can have impacts on learning

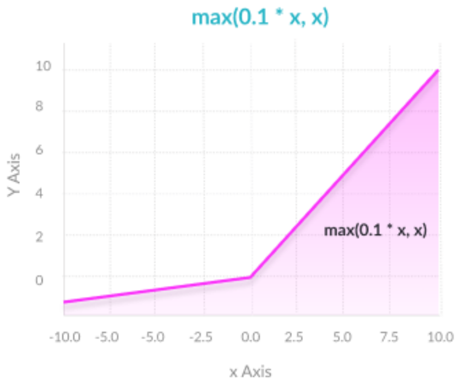
Common Activations

ReLu

- Rectified Linear Unit
  - Linear for a values greater than 0
  - 0 for negative values

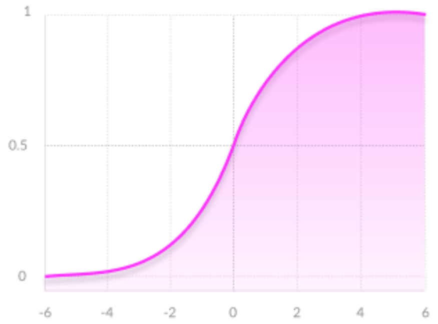
- Leaky ReLu
  - Like ReLu, but doesn't totally attenuate the values less than 0
  - Can help learning by allowing gradients to propagate with negative values



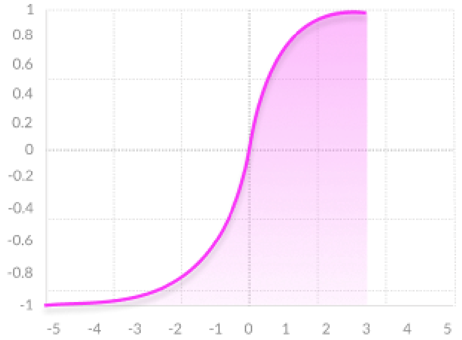


Common Activations

- Sigmoid
  - Maps input to [0, +1]
  - Acts to normalise the outputs (within fixed bounds)
  - Effectively learns a classifier



- TanH
  - Maps input [-1, +1]
  - Otherwise like the Sigmoid



Common Activations

- SoftMax Activation
  - Normalise the output such that is sums to one
  - Typically used as the output of a classification network
  - Highlight the highest response, supress all others
- There are lots of other activations
  - Exponential Linear Unit (ELU)
  - Clipped ReLu
  - SoftPlus
  - Swish

Network Layers and Computational Efficiency

- Neural Networks have a (well deserved) reputation for being computationally demanding. But operations can be implemented very efficiently.
- Consider convolution:
  - Each pixel in the output can be computed independently
  - Massive potential for parallelisation
  - Hence, rapid performance gains possible via GPUs
    - Huge numbers of very simple processing cores
- Many other operations can be similarly broken down
  - Fully connected layers are a matrix multiplication. Each output element can be computed independently of all others

CAB420: Building A Network for Classification

A SMALL ONE TO START WITH

A Network

- A collection of layers
  - Computation layers
    - Fully connected, convolution
    - Essentially can be expressed as $y=wx+b$, where all variables are matrices
  - Activation layers
    - Non-linearities between computations, regulate the flow of data
  - Pooling layers
    - Reduce dimensionality, combine activations
- Output of one layer is input to the next
  - Data propagates through the network

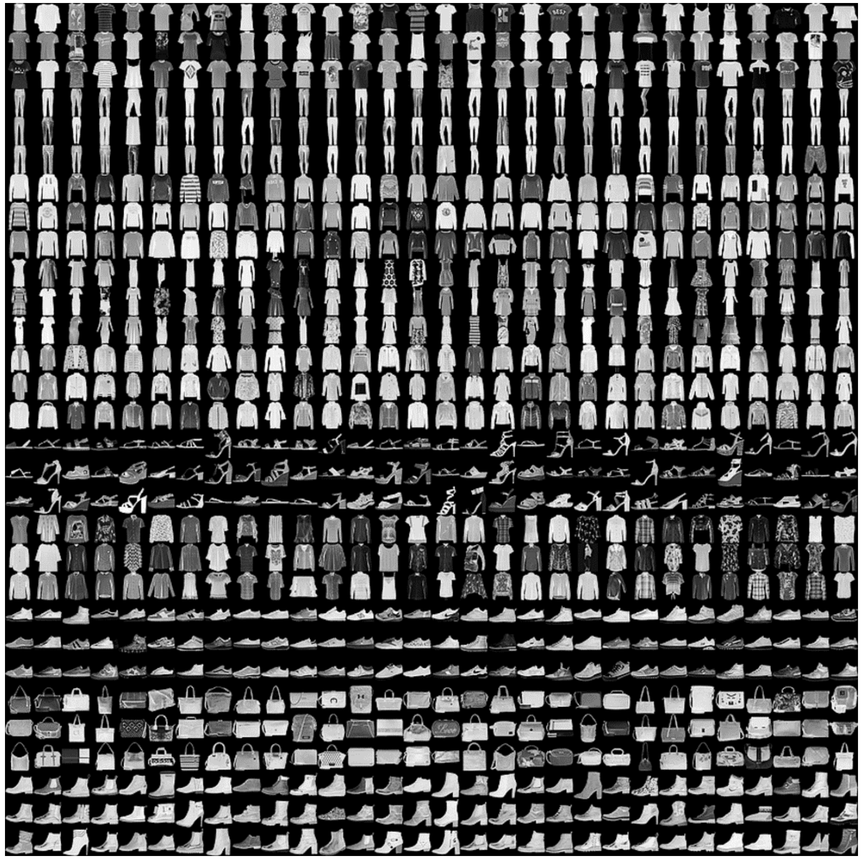
Network Structure

- Networks can have
  - Multiple branches
  - Multiple inputs and/or outputs
  - Skip connections
    - i.e. some layers are skipped and features are concatenated elsewhere
- We'll stick to simple networks (for now)
  - One input
  - One output
  - Feed-forward structure

A Classification Problem

- Fashion MNIST
  - 60,000 28x28 pixel greyscale images of clothing
  - 10 types of clothes
- The task
  - Classify images into the type of clothes they show

We're using images here. If you're uncertain about images as a data type please have a look at ***CAB420_DCNNs_Additional_Example_1_Images_Introduction.ipynb***



An Approach

- Start simple
  - A couple of fully connected layers
  - This won’t work that well

- Then add complexity
  - Convolutions!

- See ***CAB420_DCNNs_Example_1_Classification_with_Deep_Learning.ipynb***

Network Output

- We have a classification task, so our network needs to tell us which class something is. How?
  - Using a “one-hot” vector
- Consider our 10 class classification task
- We can represent this as a vector of length 10, where one element is 1 and the rest are 0’, i.e.:
  - 0001000000, would be class 4
  - 1000000000, would be class 1

Network Losses

- We need a way for our network to know when it’s right or wrong
  - Enter, the loss function
- Loss functions
  - Are 0 when the network get’s it right
  - Usually return increasingly large values as a network becomes more wrong
- These provide the errors that are back-propagated to train the network

Binary and Categorical Cross Entropy

- Use to measure the loss for classification tasks

$$CE = -\sum_{i}^{N} y_i' \log(y_i)$$

  - where
    - y_i' is the true class probability, [0..1]
    - y_i is the predicted probability, [0..1]
    - N is the total number of classes
- Measures mismatch between observed and expected distributions

Cross Entropy Explored

$$CE = -\sum_{i}^{N} y_i' \log(y_i)$$

- $y_i = [001], y_i' = [001]$
- CE = -(0xlog(0) + 0xlog(0) + 1xlog(1)) = inf +inf+0
  - Problem, log(0) is undefined
- In practice
  - Our estimates are almost never 0, activation functions see to that

Cross Entropy Explored

$$CE = -\sum_{i}^{N} y_i' \log(y_i)$$

- $y_i = [010], y_i' = [001]$
  - Note, we'll treat the 0's in y_i as very small positive numbers
- CE = -(0xlog(0.000001) + 0xlog(1) + 1xlog(0.000001)) = -(0 + 0 + -6) = 6
  - We record a high loss as our classifier was totally wrong

Cross Entropy Explored

$$CE = -\sum_{i}^{N} y_i' \log(y_i)$$

- $y_i = [0.2\ 0.4\ 0.4], y_i' = [001]$
- CE = -(0xlog(0.2) + 0xlog(0.4) + 1xlog(0.4)) = -(0 + 0 + -0.39) = 0.39
  - Our loss is not as high as we had some likelihood in the correct result

Binary vs Categorical Cross Entropy

- Categorical Cross Entropy (CCE)
  - You have N exclusive classes
- Binary Cross Entropy (BCE)
  - You have two exclusive classes
  - 2-class case of CCE
- Multi-Class Classification
  - A sample can belong to more than 1 class
  - Use BCE, effectively treat membership of class as a binary classifier

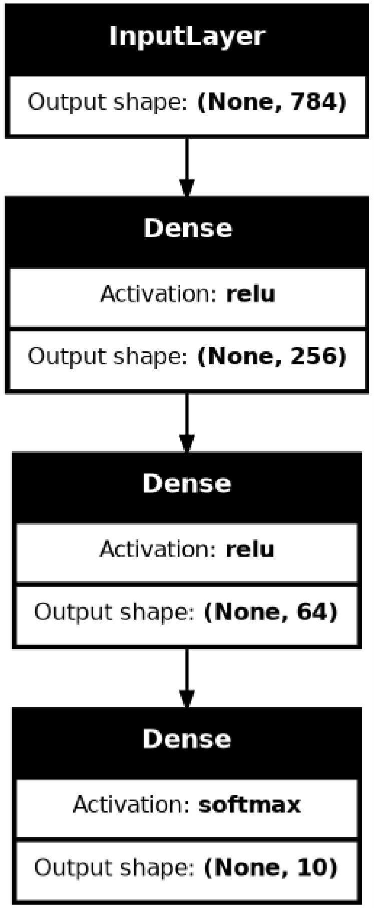
A Simple Network

- Vectorise input
  - [28 x 28] image becomes a [1 x 784] vector
    - Destroys spatial information
  - 3 dense layers
    - Intermediate 1: 256 neurons
    - Intermediate 2: 64 neurons
    - Output: 10 neurons
      - 10 classes

```
Model: "fashion_mnist_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
img (InputLayer)             [(None, 784)]             0
_________________________________________________________________
dense (Dense)                (None, 256)               200960
_________________________________________________________________
dense_1 (Dense)              (None, 64)                16448
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
=================================================================
Total params: 218,058
Trainable params: 218,058
Non-trainable params: 0
```

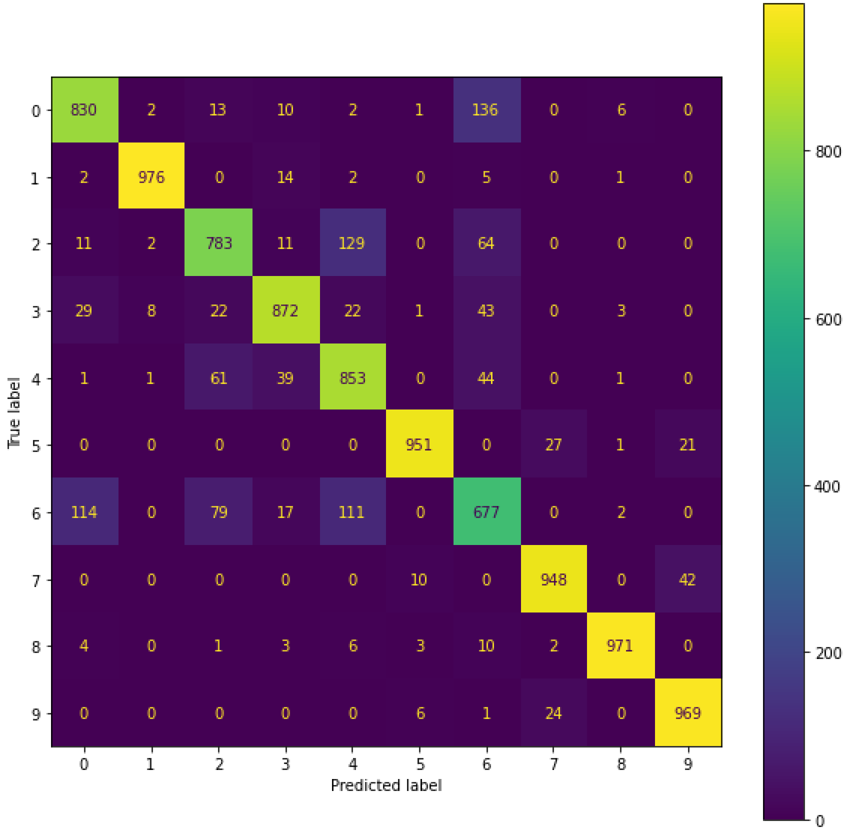
A Simple Network

- 200,000 + parameters
  - And this is a simple network
- Dense layers become smaller as we go deeper
  - Seek to discover most salient information for the task at hand
  - What is salient (important) is determined by the training



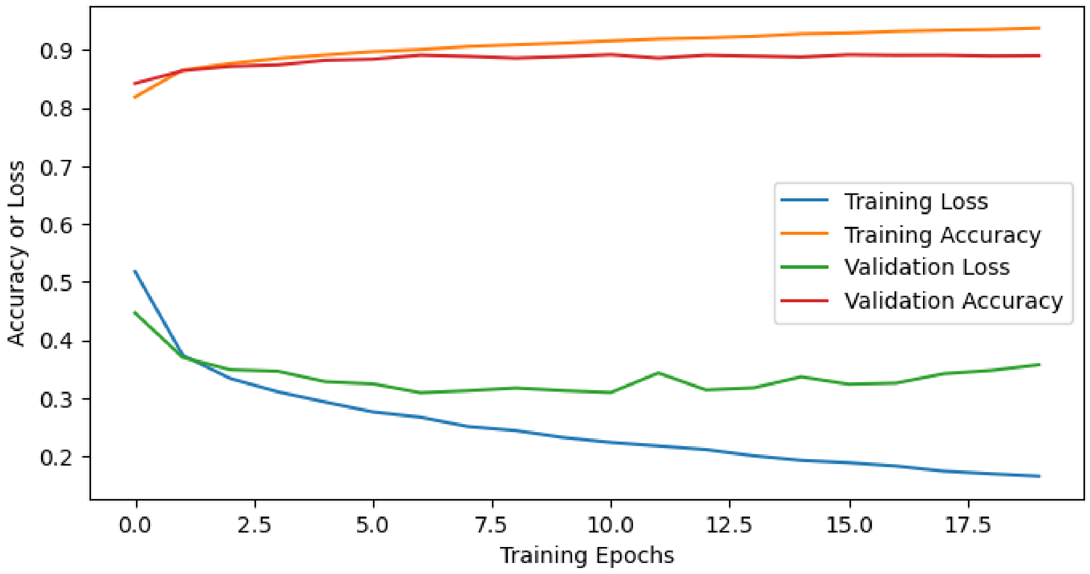
Simple Network Performance

- 88.3% accuracy on test set
- We can improve accuracy by including spatial information



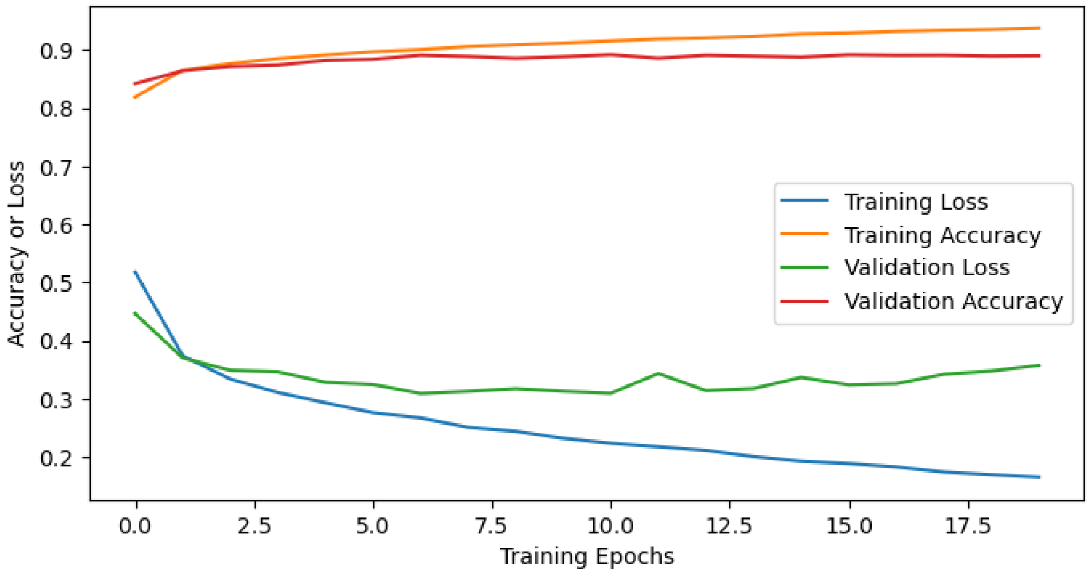
Training Performance

- This plot shows the network's performance as we train the network
  - Training Loss
    - Categorical cross entropy on the training data
      - The value of our loss
  - Training Accuracy
    - Accuracy on the training set
  - Validation Loss
    - Categorical cross entropy on the validation data
      - Unseen data, not used in training
  - Validation Accuracy
    - Accuracy on the unseen validation data



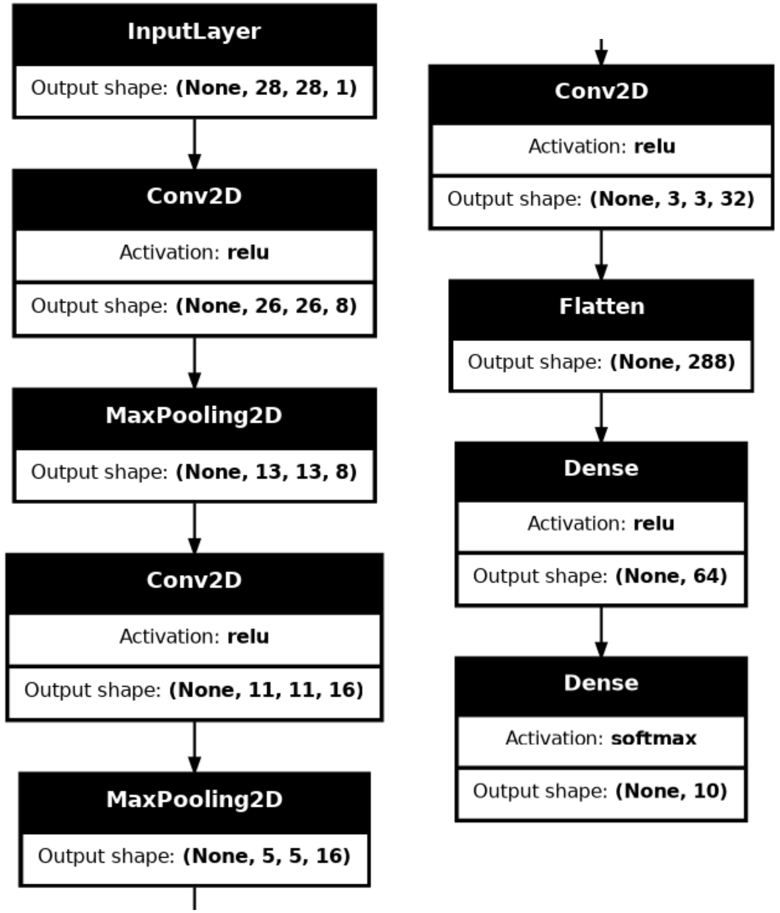
Training Performance

- In general
  - Training loss and accuracy will continue improving as we keep training
  - Validation loss and accuracy will improve with the training loss and accuracy up to some point
  - Beyond this point, the network will begin to overfit
  - If we train for too long, performance may decline on the validation set
    - This is bad, we'd like to stop training before this happens
- In our case, around 10 epochs is a good spot to stop training
  - Validation accuracy and loss have stopped improving at this point



My First CNN

- Image shaped input
  - 28 x 28 x 1
- Three 2D convolution layers
  - Max-pooling after first two
    - Reduce size of representation
    - Keep only the most important features
  - More filters as we go deeper
    - Learn simple filters on the raw image
    - Learn more complex filters over earlier outputs
  - Convolution output width and height are **not** the same as our input width and height
    - Boundary effects, we have no padding, so can't convolve at the image edges
    - Can change this as a layer parameter if we wish
- Two dense layers
  - Final layer for classification
  - Same final output structure as earlier network



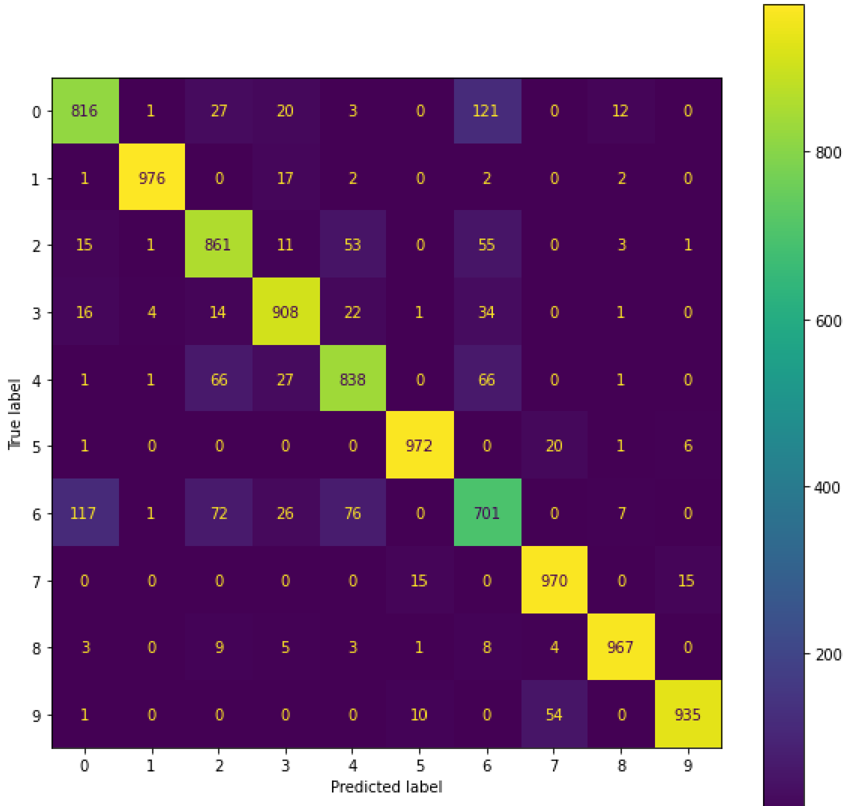
My First CNN

- 25,000 parameters
  - ~1/8th our first network
- Convolution layers are usually more efficient in terms of parameters than fully connected layers
  - 75% of our parameters are in the dense layers
- But convolution layers are more computationally intensive
  - This network will be slower to train

```
Model: "fashion_mnist_cnn_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
img (InputLayer)             [(None, 28, 28, 1)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 26, 26, 8)         80
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 8)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 16)        1168
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 16)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 3, 3, 32)          4640
_________________________________________________________________
flatten (Flatten)            (None, 288)               0
_________________________________________________________________
dense_3 (Dense)              (None, 64)                18496
_________________________________________________________________
dense_4 (Dense)              (None, 10)                650
=================================================================
Total params: 25,034
Trainable params: 25,034
Non-trainable params: 0
```

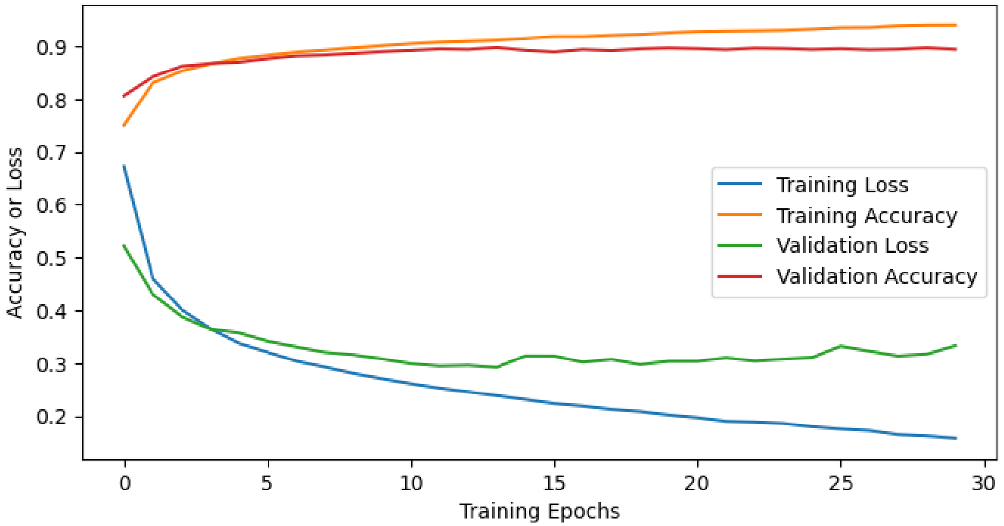
CNN Performance

- 89.4 % accuracy on testing set
  - Small improvement over dense network
  - The dense network was already quite good
    - As we get closer to perfect, it becomes harder and harder to find gains



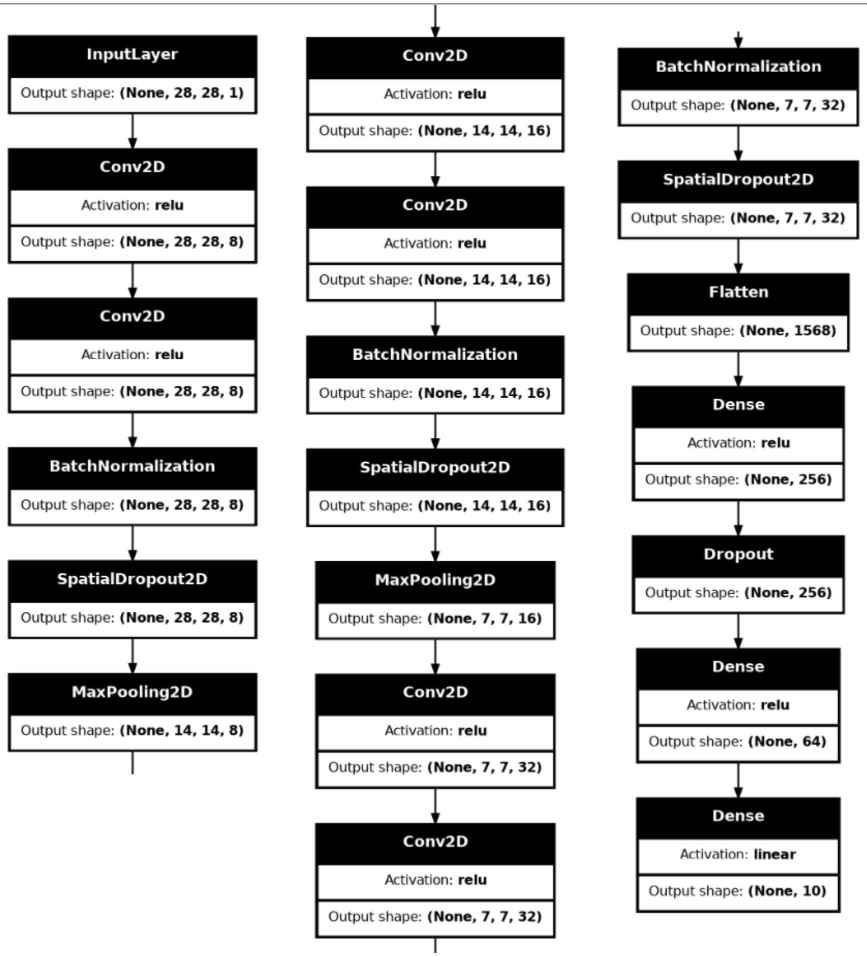
CNN Training Results

- Same shaped curves that we saw before, and the same overall performance characteristics are observed
  - Training and validation results initially improve together
  - Training results continue to improve the longer we train
  - Validation results flatten out after a while and network starts to overfit
- Network takes ~15 epochs to train
  - Point at which validation performance flattens out
  - Slightly slower than our fully connected network



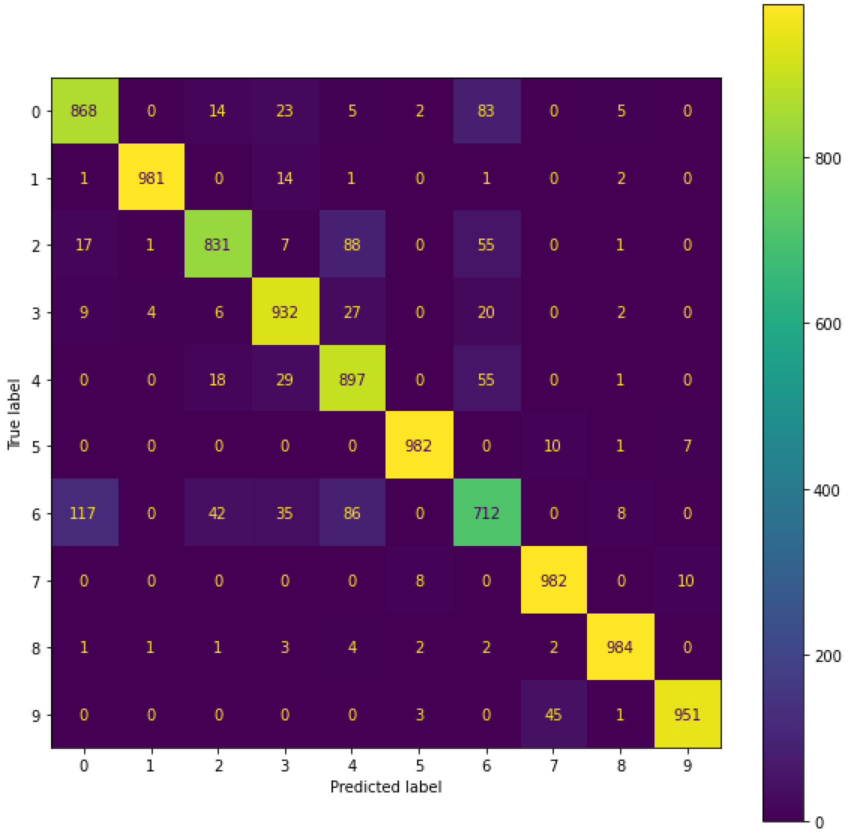
Making it Bigger

- 6 Convolution layers
  - Stacked in pairs
- 3 Dense layers
- 437,026 parameters
- Same input and output as earlier CNN
  - Just more stuff in the middle



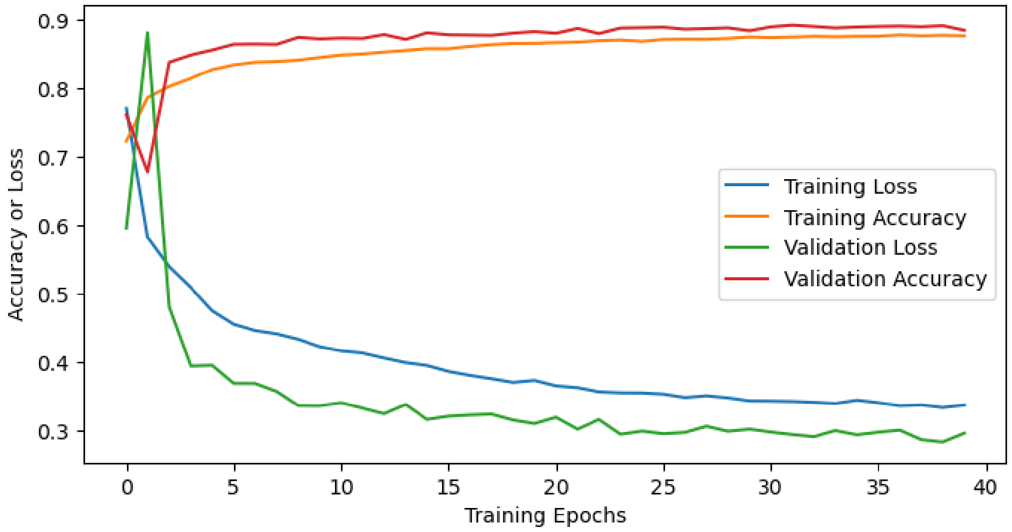
Bigger CNN Performance

- 91.2% Accuracy
- Training time has greatly increased
  - 3-4 times our earlier CNN



CNN Training Results

- Again similar, but with a twist
  - Validation accuracy and loss are ahead of training
  - Due to the dropout layer
    - More on this later
- Other broad trends still visible
  - Performance improves fast at first, then slows
  - Training continues to improve while validation tapers off
- Networks take a little longer again to train
  - Validation performance converges at around 30 epochs
  - Due to increase in complexity



Network Size and Accuracy and CAB420

- To a point, larger more complex networks will give better performance, however
  - Gains decrease as networks grow
  - Larger network take longer to train, and require more memory
  - Larger networks need more data and are more likely to overfit
  - We can go too deep and break things

- In CAB420, we are not expecting you to train models for hours at a time
- When playing with networks
  - Start small
  - Accept that you will not get state of the art performance
  - Consider using services such as the QUT hosted Jupyter Notebook, or Google Colab to access GPUs if you don't have one

CAB420: Regression with Deep Nets

A LOT LIKE CLASSIFICATION WITH DEEP NETS

Regression with Deep Nets

- As simple as changing our output layer
  - For classification, we have a “softmax” output
    - 0 or 1 (or somewhere in between) to indicate classification certainty
  - For regression, we want a continuous output (usually)
    - ReLu (or similar) activation
    - And a regression target to learn against
  - Can regress to multiple outputs

- Other than that, the networks are pretty similar

Regression Losses

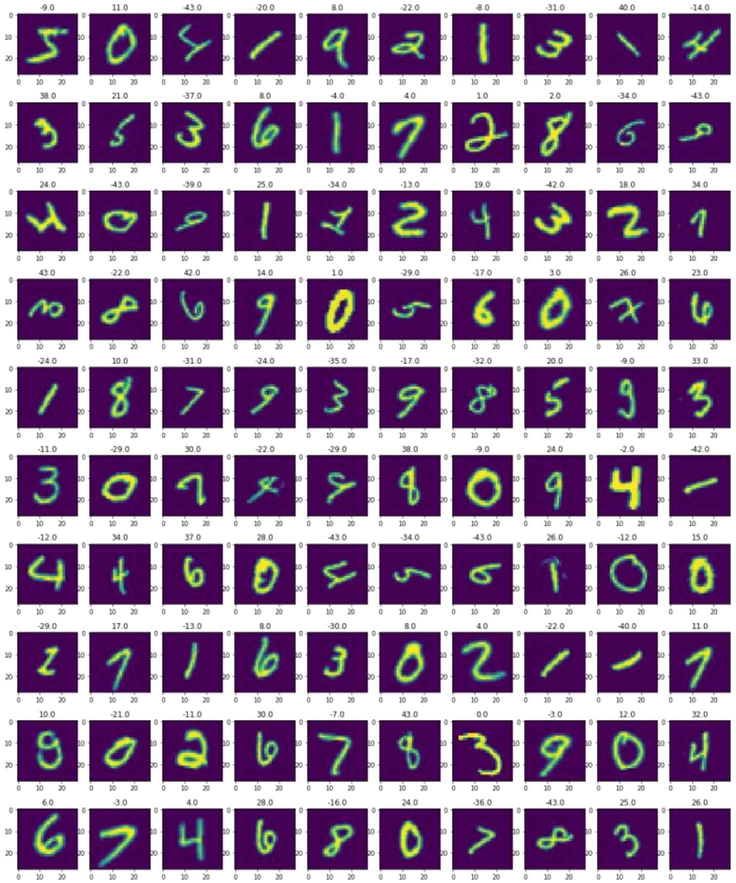
- Usually, we'll use something like Mean Squared Error

$$\text{MSE} = \sum_{i}^{N} (y_i' - y_i)^2$$

- Other times we may wish to use Mean Absolute Error, or other distance measures depending on the problem and data
  - You can see a list of existing losses within tensorflow/keras here: https://keras.io/api/losses/

Regression Example

- See ***CAB420_DCNNs_Example_2_Regression_with_Deep_Learning.ipynb***
- Data
  - Rotated digits, digits have been randomly rotated by [-45 … +45] degrees
- Task
  - Estimate the amount of rotation a digit has undergone
    - Single output, regressing from an input to one number



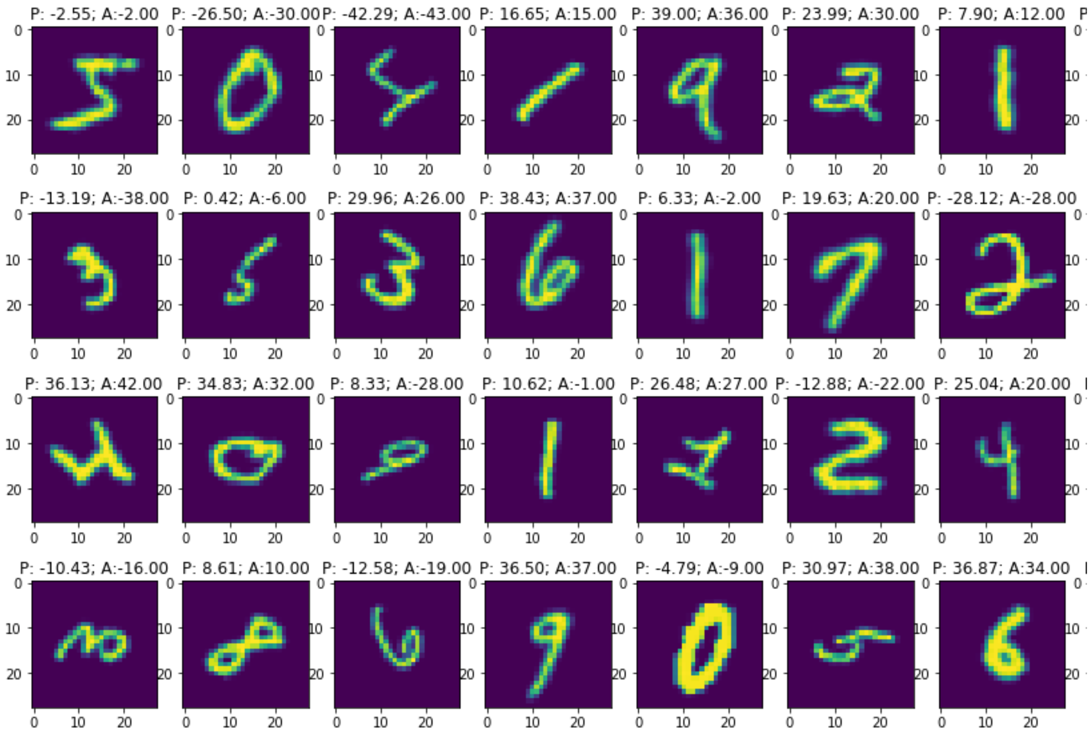
The Network

- Almost identical to "My First CNN"
  - CNN architectures are very adaptable
- One change
  - Our final dense layer is now size 1
- We also change our loss
  - MSE rather than categorical cross entropy
  - Could also use MAE (mean absolute error), or other regression loss

```
Model: "mnist_angles_cnn_model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
img (InputLayer)             [(None, 28, 28, 1)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 26, 26, 8)         80
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 13, 13, 8)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 11, 11, 16)        1168
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 16)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 3, 3, 32)          4640
_________________________________________________________________
flatten (Flatten)            (None, 288)               0
_________________________________________________________________
dense (Dense)                (None, 64)                18496
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 65
=================================================================
Total params: 24,449
Trainable params: 24,449
Non-trainable params: 0
```

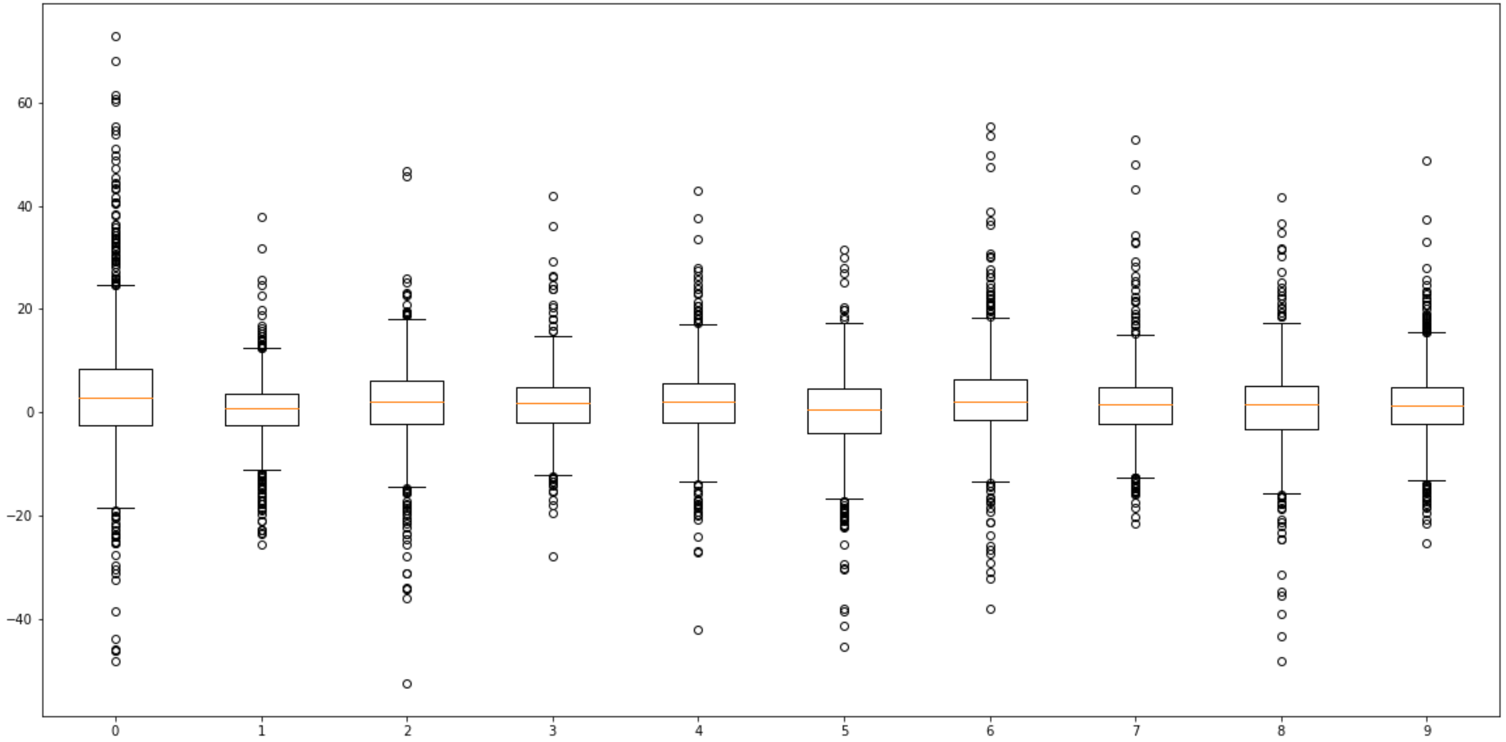
Results

- Model is fairly accurate
  - Can estimate rotation for all digits
- Network has no explicit knowledge of the digits



Results

- Network finds 0 the hardest to correct
  - Performance broadly similar for all digits though



CAB420: What is Learned?

PARAMETERS, LOTS OF PARAMETERS

DCNNs as a “Black Box”

- You will often see DCNNs and Deep Learning models referred to as a “Black Box”
  - Based on the idea that such models are hard to (or even impossible to) understand or interpret

- Consider a linear regressor
  - Learned parameters are the values of β
  - β define the importance and direction of influence for each variable

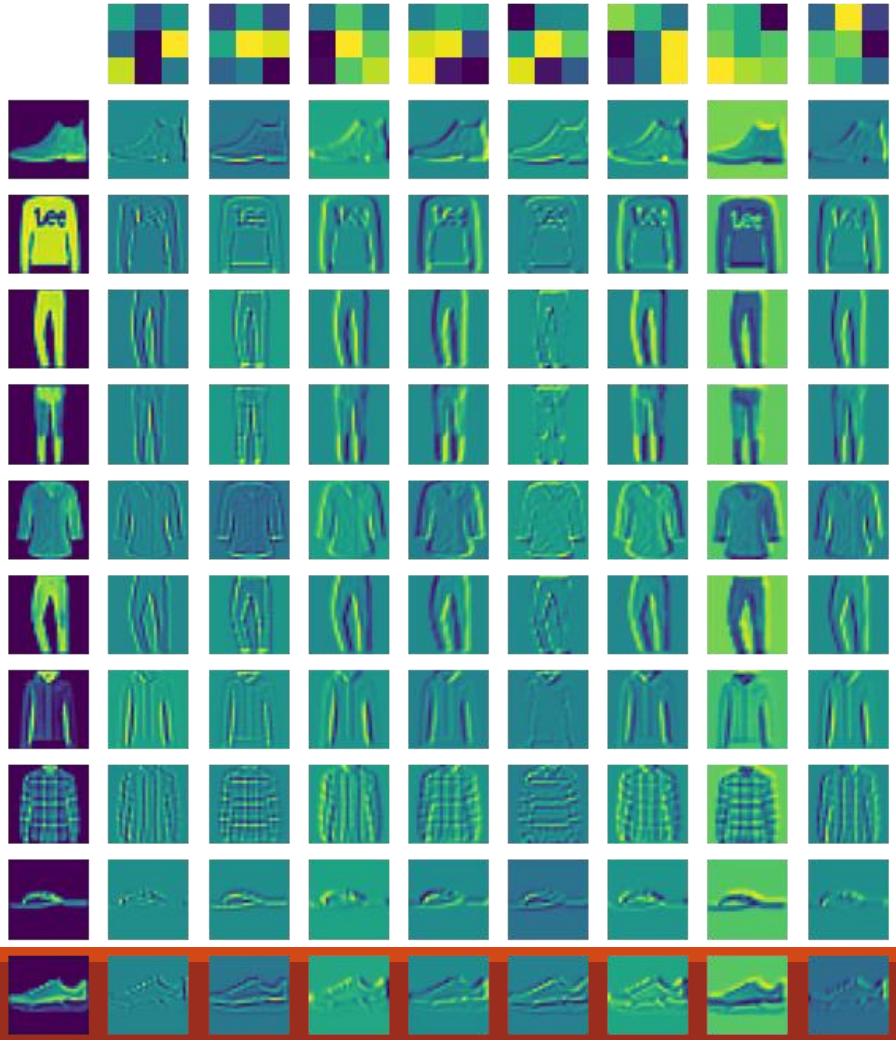
- For a DCNN the “meaning” of the learned parameters is less obvious
  - We can still access them all, but there are many, many more of them, which is really the problem
  - But we can access and visualize parameters if we wish
  - See ***CAB420_DCNNs_Additional_Example_3_What_Does_the_Network_Learn.ipynb***

Dense Layers

- For a dense layer we have:
  - An input vector, x, of length M
  - An output vector, y, of length N
- The dense layer learns
  - $y = wx + b$
  - where w is a matrix of size $M \times N$, and b is a bias vector of size N
- The dense layer operation can be decomposed as follows:
  - $y[0] = w[:,0]x + b[0]$
  - $y[1] = w[:,1]x + b[1]$
  - …
  - $y[N-1] = w[:,N-1]x + b[N-1]$
- Each of the above lines is a single linear regressor
  - The values in w simply indicate the strength and direction of influence

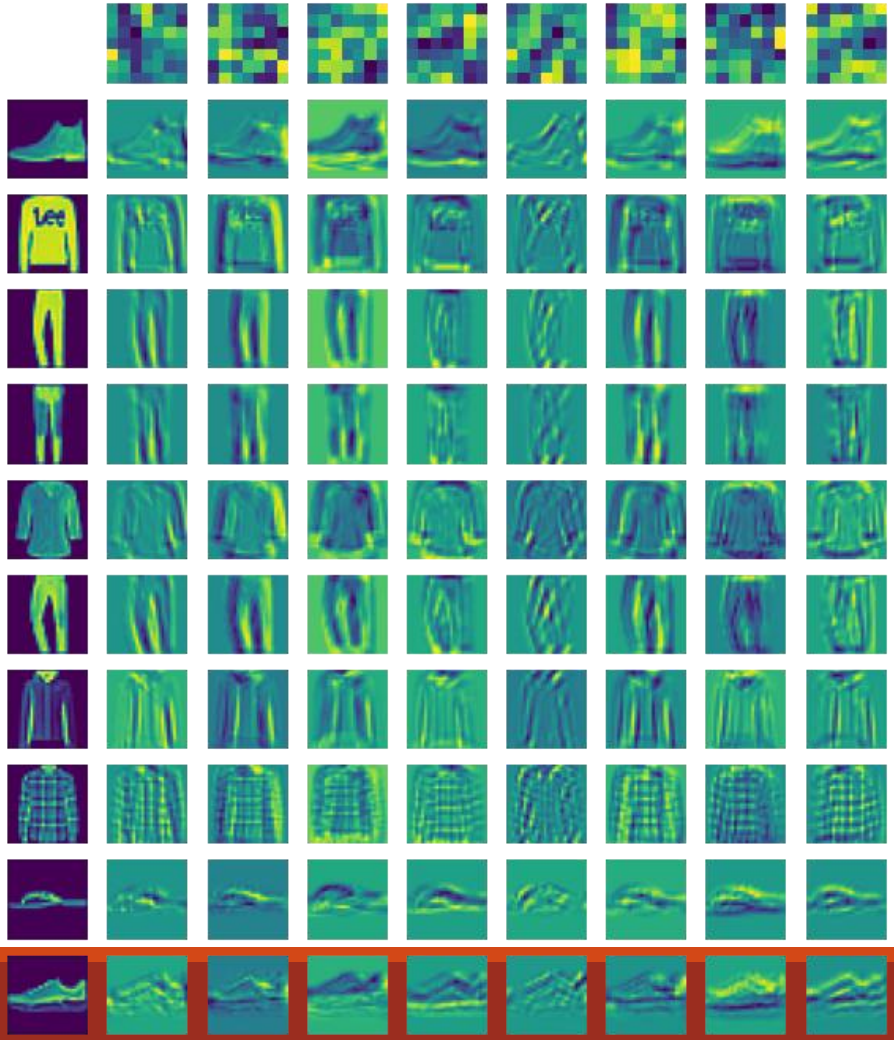
Convolution Layers

- Convolution layers learn a set of filters
  - We can visualise the filters, and their output (response)
- Left column: Input images
- Top row: Learned 3x3 filters
  - First convolutional layer
- The rest: Responses to filters
  - Notes that each image has been scaled independently
- Filters focus on edges
  - Different filters capture different edges
- Filters are unique
  - But limited 3x3 patterns are possible



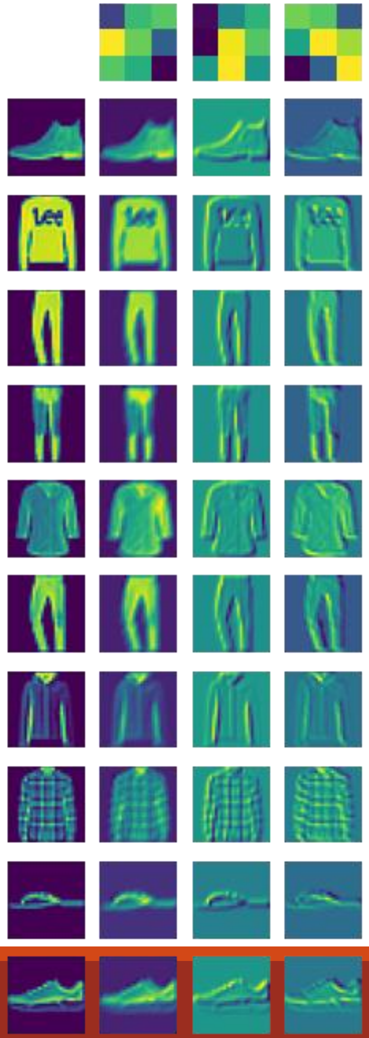
Larger Convolution Filters

- Same setup as before, but with 7x7 filters
  - More complex filters leads to more complex response maps
  - Larger filters consider a larger area of the input image
- Many more possible 7x7 patterns than 3x3



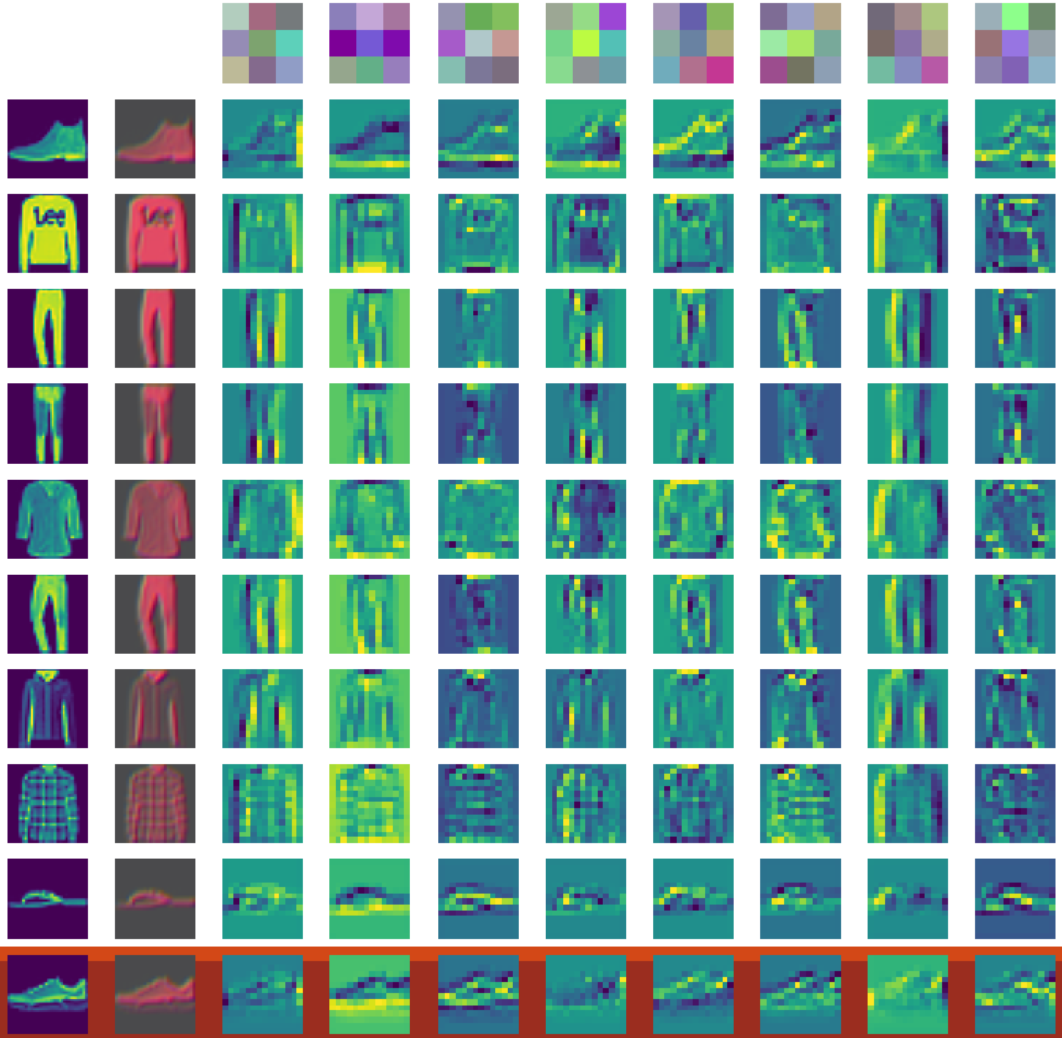
Stacking Convolution Filters

- Two convolution layers
- First layer has three 3x3 filters
  - Response maps similar to what we saw before, just less of them
  - Detecting edges in different orientations



Stacking Convolution Filters

- Left column: Input Images
- 2nd column from left: Output of first layer
- Second layer has 8 3x3 filters
- Second layer's input is the stacked responses from the first layer
  - First layer has three filters, first layer output is a three channel image
  - Second layer learns 3x3x3 filters
  - There is also a max-pooling between the two layers
- Filter responses much more complex than first layer
  - Similar in complexity to 7x7 filters
  - Filters are looking for interactions between outputs of the first layer



Interpreting DCNNs

- The challenge is not that we can't get to the parameters, it's that there's so many of them
- In practice
  - We will rarely, if ever, visualise learned weights (i.e. the filter kernels)
  - We will look at intermediate outputs at times
    - Visualise filter responses to understand what the network is looking at
  - We use other techniques to understand what a network is looking at
    - Class Activation Maps (CAM) and Gradient-weighted Class Activation Maps (Grad-CAM)
      - Indicate what regions of image contribute to the score for a class
    - Shapely Values
      - Indicate the contribution of each input dimension to a decision
    - Neural Conductance
      - Captures information flow through the neural network
    - And many, many, more
- Interpreting DCNNs is outside the scope of CAB420, but it's important to be aware that these methods exist
  - But if you are interested, see the bonus examples

CAB420: Training Your Network

CAUSE WE KIND OF IGNORED THAT BEFORE

Back Propagation and Gradient Descent

- Neural Networks are trained using Back Propagation and Gradient Descent
  - Change the weight and bias terms using gradient descent
  - Can have problems when
    - Gradients becomes very small (vanishing gradients)
    - Gradients become very big (exploding gradients
  - Partial derivatives are used to update parameters
    - Becomes very complex, for large networks
    - Occurs behind the scenes in CAB420
  - Back propagation is a crucial component of neural networks that allows for optimisation of the cost function.
  - Gradient descent allows us to approach an optimal solution over a number of iterations

Gradient Descent

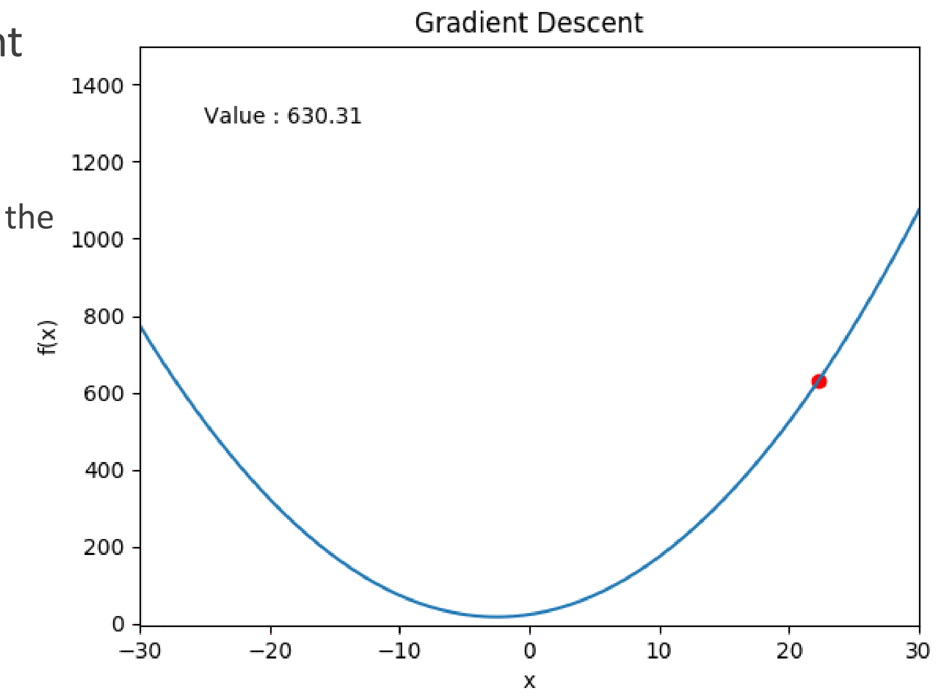
ROLLING BALLS DOWN HILLS

Optimisation

- Usually in machine learning we can’t directly determine a model’s parameters
  - i.e. we can’t directly determine model coefficients
- In such cases we can use an iterative approach:
  - Make an initial estimate
  - Evaluate that estimate
  - Update the estimate and evaluate again
  - Repeat until either
    - The estimate stops changing (or only changes slightly)
    - A maximum number of steps is reached

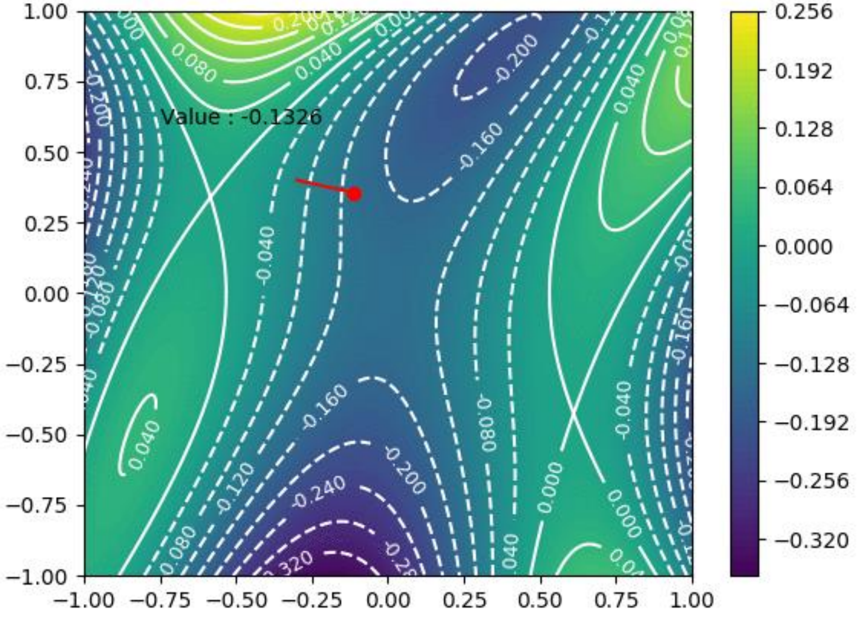
Gradient Descent

- One of the most popular optimisers is Gradient Descent
  - Start at some estimate
  - Evaluate the gradient
  - Move in a direction that minimises the gradient



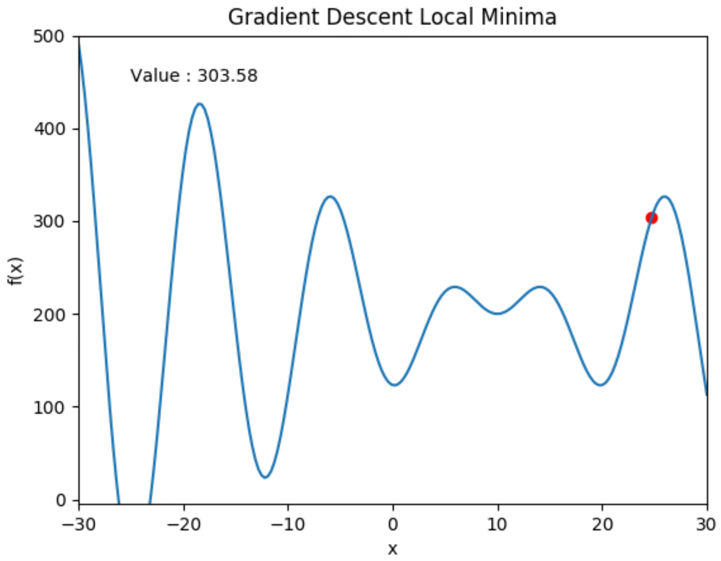
Gradient Descent

- Scales to an arbitrary number of dimensions
  - Uses partial derivatives to determine gradients



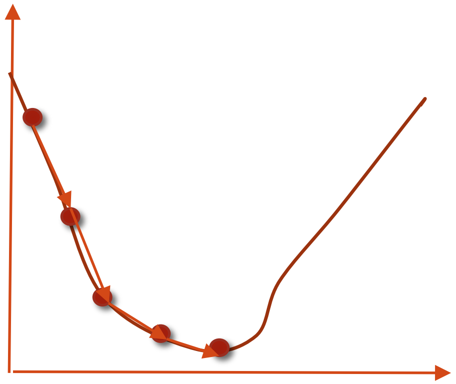
Gradient Descent

- Sensitive to starting conditions
  - Can get stuck in local minima rather than finding global minima

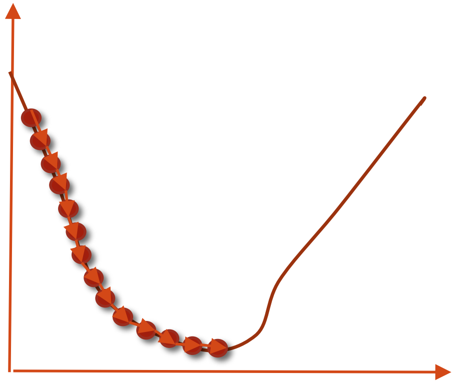


Gradient Descent

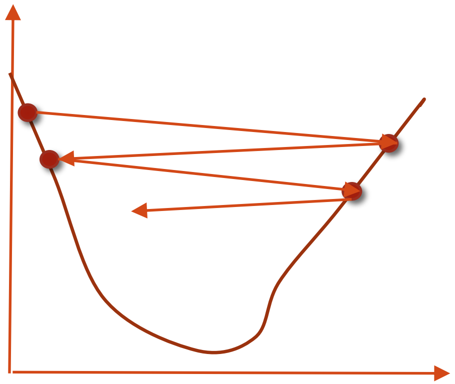
- Learning rate is important
  - How much do you change the model each step?
- Too slow
  - Takes a long time to get to a solution
    - More prone to getting stuck in local minima
- Too big
  - Can “jump over” the best solution



Good Learning Rate



Slow Learning Rate



Fast Learning Rate

Training DCNNs

BY ROLLING BALLS DOWN HILLS

Terminology

- Epoch
  - One complete pass through the data
  - After one epoch, the network has seen all examples
- Batch
  - One update of the networks, based on a small sample of data
- Optimiser
  - Gradient descent approach that we use to train
- Learning Rate
  - How fast we allow the model parameters to update

Why not train on all data at once?

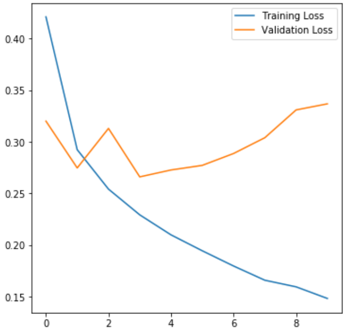
- Consider Fashion MNIST, we have
  - 28x28x50,000 = 39,200,000 pixels
  - That's a lot to process at once
    - And this is a "toy" dataset
- For most tasks, parsing all data at once is not practical
  - Hence, batches
- A batch is a smallish collection of inputs
  - Usually somewhere between 1-256 depending on
    - How much data you have
    - How big the network is
    - How much money you spent on hardware (or how much you can fit in memory)

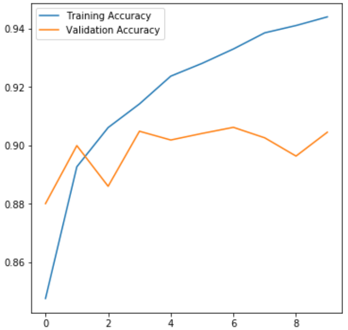
Batch Size vs Epochs

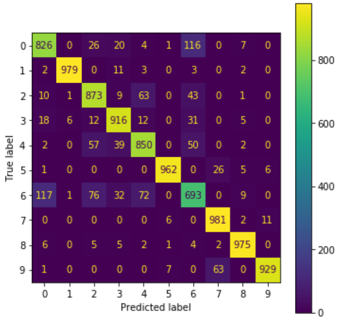
- A small batch size means
  - More updates per epoch
  - Can train the network in fewer epochs because you have more updates
  - But….
    - Each batch is less representative of the overall data shape
    - Can lead to a poor fit depending on how imbalanced data is

Impacts of Batch Size

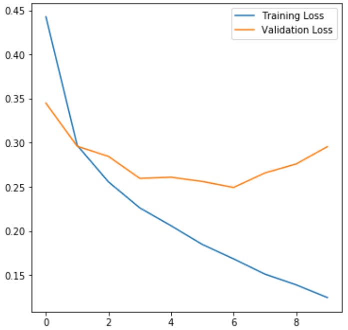
- See ***CAB420_DCNNs_Additional_Example_4_Training_Parameters.ipynb***
- Batch size = 4

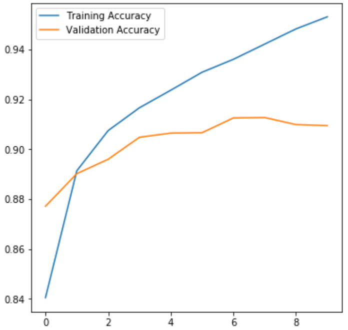


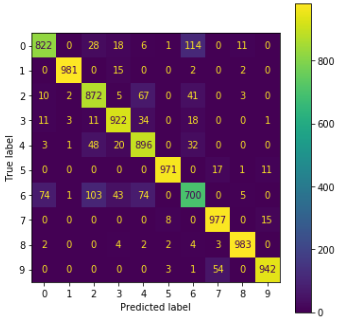




- Batch size = 16

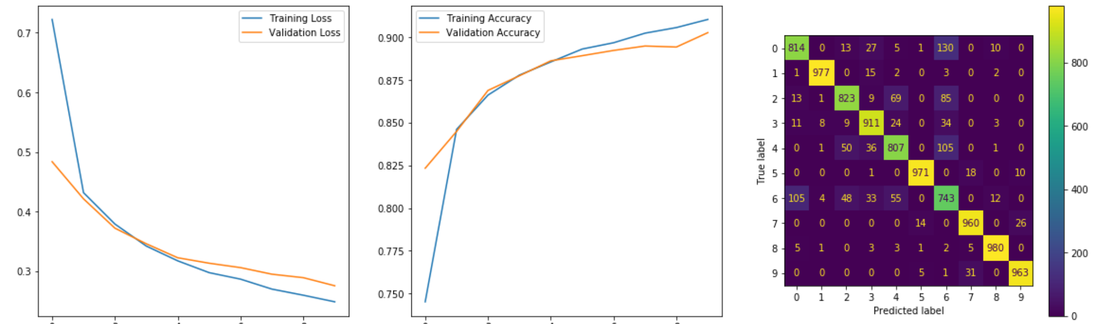




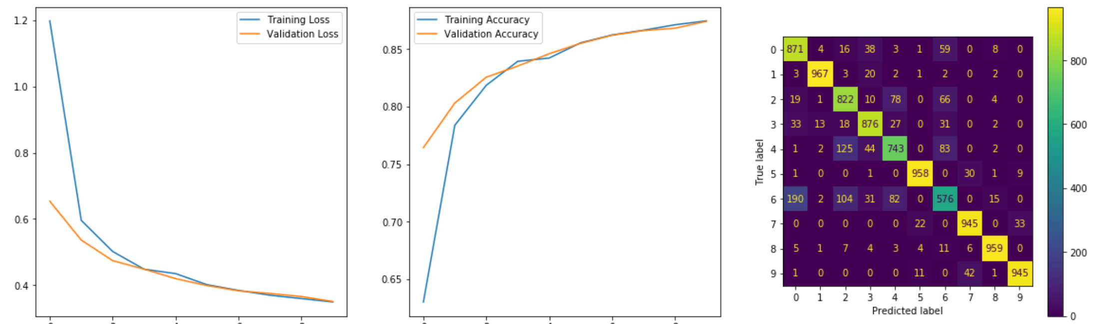


Impacts of Batch Size

- Batch size = 256



- Batch size = 1024



- Larger batch size leads to
  - Smoother training
  - Slower convergence (in terms of epochs)
  - Higher memory requirements

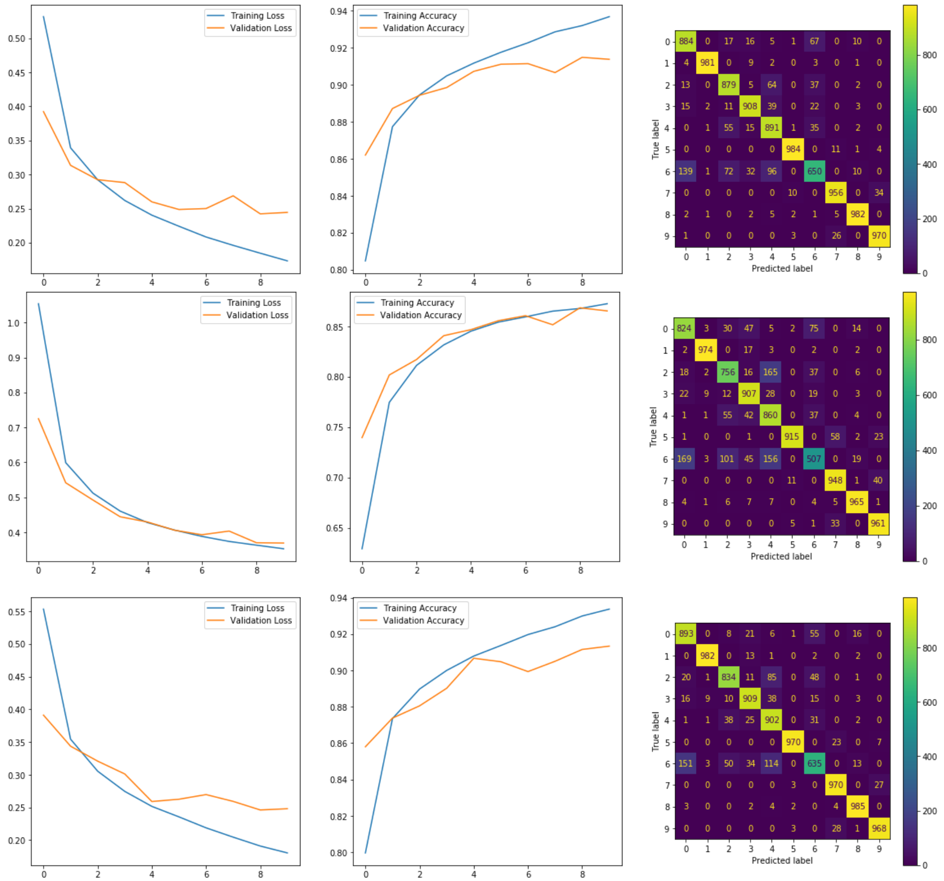
Optimisers

- All based on gradient descent to train via backpropagation
  - Propagate gradients back up the network to adjust weights

- Many options exist
  - There is no real “standard” optimiser
  - Adam is the closest thing to a default
  - Differences between optimisers are often small (see https://arxiv.org/abs/2007.01547), and the variation in performance for a single optimiser is often larger than the difference between optimisers
  - Some tasks or networks will work better a given optimiser
    - This is not consistent however

Optimisers

- Three training runs
  - RMSProp (Top)
  - SGD (Middle)
  - Adam (Bottom)
- SGD slower to learn
- All achieve similar performance
  - Our model and data are not complex and so similar performance is achieved for all
  - If you re-run this, you will see some variation. Maybe SDG will be a bit quicker next time.
  - You will likely see minimal variation in CAB420 with regards to optimiser choice
  - If in doubt, use Adam

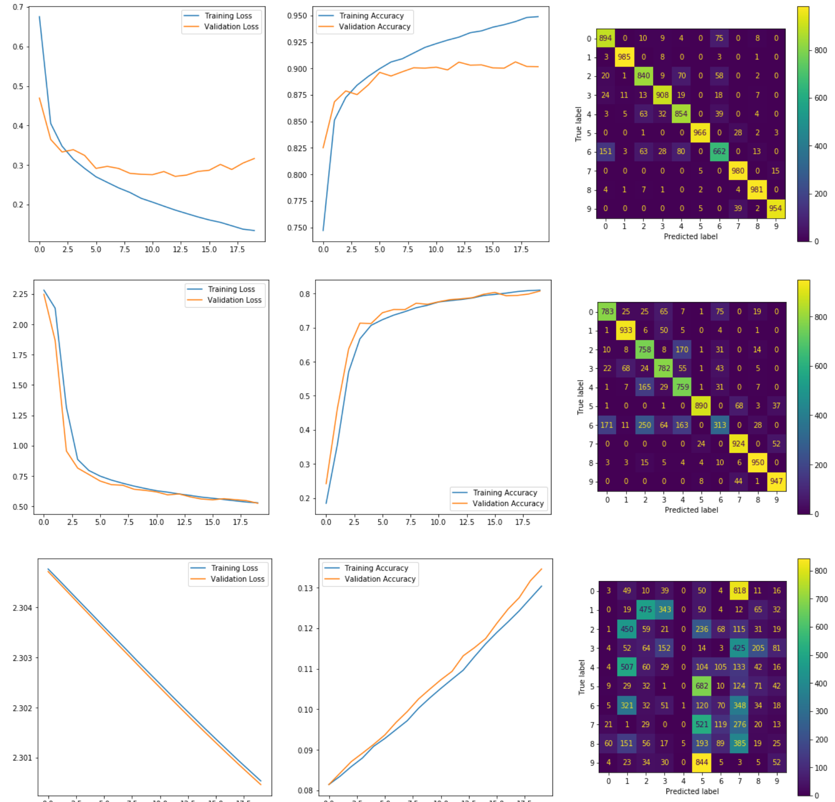


Learning Rate

- Bigger number -> Faster Learning
- Faster learning can be good early
  - You’re a long way from a solution
- Slower learning is better once you have a good estimate
  - With fast learning, the danger is you “overshoot” the solution
- A good practice is to use a learning rate schedule
  - For example, start fast, drop by a factor of 10 every 10 epochs

Learning Rate

- Fast (0.1)
  - Converged after ~8 epochs
- Slow (0.001)
  - Almost converged after 20 epochs
- Glacial (0.00001)
  - Nowhere near convergence after 20 epochs



Avoiding Overfitting

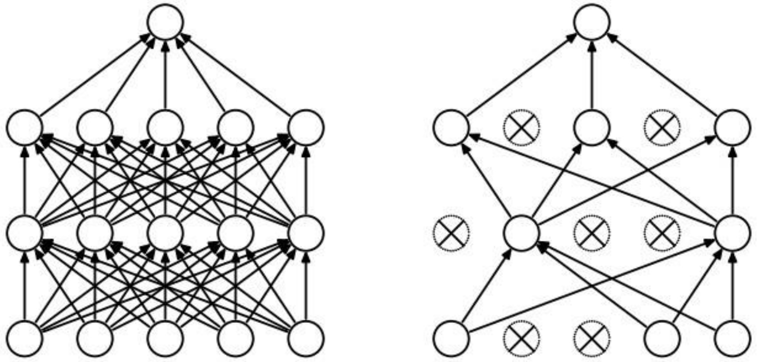
AND OTHER TRICKS AND HACKS

Overfitting with DeepNets

- It’s really easy to overfit
  - Potentially millions of parameters
  - Almost always have more parameters than samples
- Two possible approaches
  - Modify the network to reduce overfitting chance
  - Get more data
    - Or make it up

Drop Out

- Randomly disconnect a portion of neurons each pass through the network
- Why?
  - Means we never learn on the whole network at once
  - Reduces overfitting
  - But slows training
- Can be applied at different levels
  - At neurons
  - At Convolutional Filters
    - Spatial Dropout, drops a percentage of whole filters


(a) Standard Neural Net (b) After applying dropout.

Batch Normalisation

- Neural networks propagate information from one layer to the next
  - Layer N takes results of Layer N-1 as input
    - And N-1 takes results of N-2, and so on
  - What if the range of values coming from N-1 keeps changing?
    - This tends to happen a lot during the early stages of training
- Batch Normalisation helps address this
  - Improves training speed
  - Reduces overfitting

Batch Normalisation

- Batch normalisation normalises the output of a batch at a designated point in the network
  - By default, 0 mean and unit std.dev
    - But can learn a different mean and std.dev
- Why?
  - If we perform batch norm after layer N-1, we now know that the input to Layer N will have 0 mean and unit std.dev
  - Makes it easier to learn layer N as the layer will always get data in the same range
  - Essentially provides a model checkpoint

Batch Normalisation

- Layer placement impacts performance
  - Generally, place before an activation
  - BatchNorm will standardise outputs around a learned mean
    - If placed after an activation, outputs have been altered by the activation
    - Placing before an activation makes it easier to consider the impact of the proceeding layer
- Not needed after every layer
  - Consider adding after repeating blocks
    - i.e. after pairs of convolutions
  - Experiment with placement

Weight Regularisation

- Neural networks have lots of weights
  - Big weights can indicate overfitting
  - Much like Ridge regression, we'd prefer smaller weights
- Weight Regularisation applies a penalty to the network based on the total sum of the weights
  - Can be L2 (like Ridge regression)
  - Or L1 (like Lasso)
  - Or a combination
- In Keras/Tensorflow
  - Specified per layer on any (or all) of
    - Weights (kernal)
    - Bias
    - Activation
  - Flexible in terms of regulariser (L1, L2, L1 and L2, custom) used
  - Off on all layers by default

Demo Script

- See ***CAB420_DCNNs_Additional_Example_4_Layer_Order_and_Overfitting.ipynb***
  - Explore in your own time
    - Don’t feel you need to understand everything immediately
    - Play with the options in here over time
  - Feel free to ask questions, and try things out in different examples

DCNNs and Variation

YMMV

Network Initialisation, Training and Randomness

- There is lots of randomisation in a neural network training process
  - Network parameters (weights and biases) are randomly initialised
  - Data is randomly shuffled after each epoch
  - Training and validation splits may be random

- Unless controlled for, no two training runs are exactly the same
  - If you're training to convergence (or close to), they will be similar

- This is why you may have noticed variation between what's in the Git repository, and what's in some slides

An Experiment

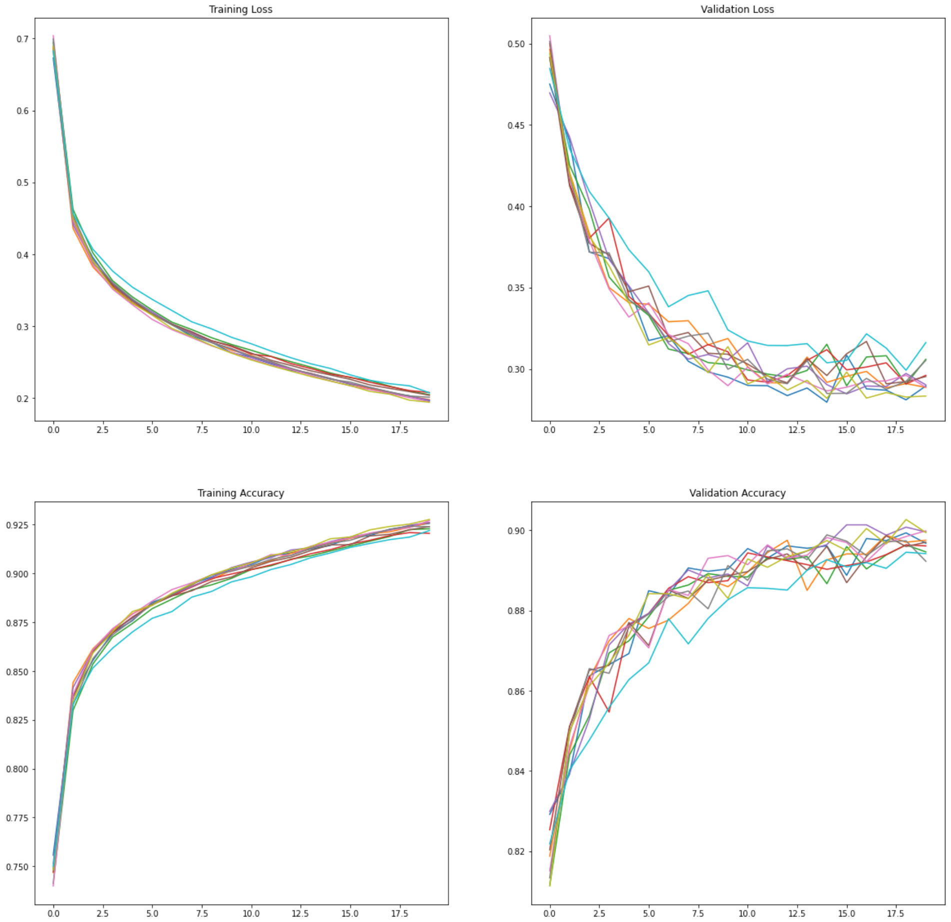
- See ***CAB420_DCNNs_Additional_Example_5_Variation.ipynb***

- 10 identical simple CNNs
- All trained on Fashion MNIST for the same length of time
- Same batch size, same optimiser
- Seek to see what sort of variation is observed in the models

10 Models

- Results are all similar
  - But there is variation
- More variation in validation performance than testing performance

- A note on convergence
  - Validation loss and accuracy curves have flattened out in these plots
  - Training much beyond the 20 epochs leads to overfitting
  - Training accuracy will continue to improve towards 100% if training continues



Differences in Predictions

- Let's consider some misclassified examples
  - Examples are misclassified by the first model
  - Report results for all models and the ground truth

- Different models (sometimes) make different decisions
  - Models can vary in that some may be right and wrong
  - Others may have different wrong answers
  - Even when all models make the same prediction, the SoftMax value varies

```
Index 17; True Class: 4
             Model 1, Predicted class 6 (0.708114)
             Model 2, Predicted class 4 (0.970968)
             Model 3, Predicted class 2 (0.649878)
             Model 4, Predicted class 2 (0.569351)
             Model 5, Predicted class 6 (0.894694)
             Model 6, Predicted class 4 (0.583097)
             Model 7, Predicted class 4 (0.917198)
             Model 8, Predicted class 4 (0.984307)
             Model 9, Predicted class 4 (0.945560)
             Model 10, Predicted class 2 (0.624152)
             Average Model, Predicted class 4 (0.506741)
Index 23; True Class: 9
             Model 1, Predicted class 5 (0.999711)
             Model 2, Predicted class 5 (0.999991)
             Model 3, Predicted class 5 (0.999992)
             Model 4, Predicted class 5 (0.998271)
             Model 5, Predicted class 5 (1.000000)
             Model 6, Predicted class 5 (1.000000)
             Model 7, Predicted class 5 (0.999940)
             Model 8, Predicted class 5 (0.840722)
             Model 9, Predicted class 5 (0.999996)
             Model 10, Predicted class 5 (0.999162)
             Average Model, Predicted class 5 (0.983778)
```

Averaging Models

- We can create an ***ensemble*** of models
  - Average the results of a set of identical models
  - “The wisdom of the crowds” for deep nets
  - Similar to Random Forests
    - Though each “tree” is a bit more complex
- Original models all perform at around 89% accuracy
  - Model 1 achieves 89.45%
- Ensemble achieves 91.53%
  - Small, but noticeable gain
- Is this worth it?
  - 2% performance gain for 10x the compute
  - However we see diminishing returns as we increase complexity anyway. Is this much worse?
  - This has the added benefit of giving us a way to measure confidence
    - This is otherwise difficult with deep networks