Deep Learning

Contents page

Topic one | Topic two | Topic three | Topic four | Topic five

L1 Machine Learning to Deep Learning

* What is deep learning

Deep learning has evolved into a central tool for solving perception problems e.g.
- recognising whats in an image
- understanding conversations
- helping robot explore the world and interact with it

It is the state of the art on everything to do with computer vision and speech recognition

Applications are increasing to
- Medicine discovery
- Understanding natural language
- Understanding documents for search ranking

Lots of data + complex problem → deep learning!

* Course overview

Lesson 1 - Logistic Classification
- Stochastic Optimisation
- Data & Parameter Tuning

Train first simple model

Lesson 2 - Deep Networks
- Regularisation

← for larger problems

Lesson 3 - Convolutional Networks ← images

Lesson 4 - Embeddings
- Recurrent Models

Text & Sequences

Assignments - Tensor flow & iPython Notebooks

* Neural Networks

1980's	Fukushima's Neocognitron
1990's	Le Cun's Lenet 5
2000	
2010's	Krizhevsky's Alexnet

↓ slow computers

2009 Speech Recognition
2012 Computer Vision
2014 Machine Translation

GPUs! Thank you gamers

* Supervised Classification

* Logistic Classification

Linear model:

$$WX + b = Y$$

W: weights, X: input, b: bias, Y: output (Trained)

$Y \rightarrow$ scores or logits

SOFTMAX function: $S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_i}}$ → Probability of being in each class

Let's look at softmax (see ipython notebook)

$$Y = \begin{pmatrix} 3 \\ 1 \\ 0.2 \end{pmatrix} \Rightarrow S(Y) = \begin{pmatrix} 0.84 \\ 0.11 \\ 0.05 \end{pmatrix} \qquad S(10Y) = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \qquad S\left(\frac{Y}{10}\right) = \begin{pmatrix} 0.39 \\ 0.32 \\ 0.29 \end{pmatrix}$$

$S(10Y)$: zeros and ones ⇒ greater certainty

$S(Y/10)$: uniformly distributed ⇒ greater uncertainty

We want our classifier to increase the certainty with which it makes predictions over time i.e. we want our logits to grow.

* One Hot Encoding

The case where we predict the correct class with probability one and all other classes with probability zero is described as 'one hot encoding'.

Clearly there are as many possible one hot encodings as there are classes.

One hot encoding works well for most problems but runs into issues when there are a large (10K / 1M) number of classes. In this case the vectors are large and mostly zeros which is very inefficient. Embedding is an approach to deal with this.

* Cross Entropy

We can measure how good our prediction is by comparing it to the corresponding one hot encoding using cross entropy:

$$\text{cross entropy} = D(S, L) = -\sum_i L_i \log(S_i)$$

(S: logits; L: one hot encoding)

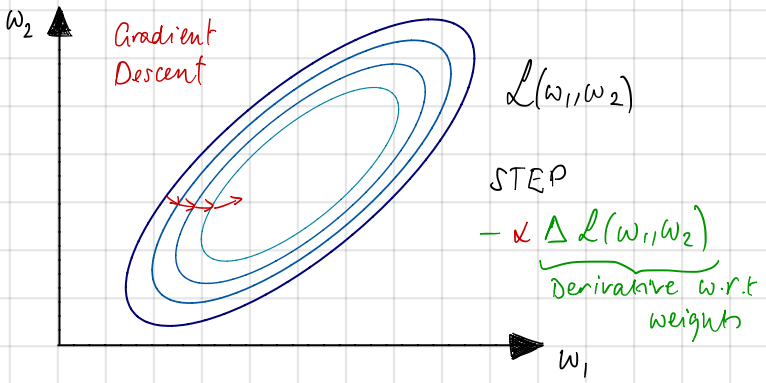
e.g. $S(Y) = \begin{pmatrix} 0.7 \\ 0.2 \\ 0.1 \end{pmatrix} \quad L = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$

This is Multinomial Logistic Classification $\rightarrow D(S(WX+b), L)$

To find the weights and bias, we minimise the Loss $\mathcal{L}$ or average cross entropy

$$\mathcal{L} = \frac{1}{N} \sum_i D(S(WX_i + b), L_i)$$

So the problem becomes one of optimisation



Later – Tools for finding derivatives
– Positives and negatives of gradient descent

First - How to feed image pixels to the classifier?
- Where do we initialise the optimisation?

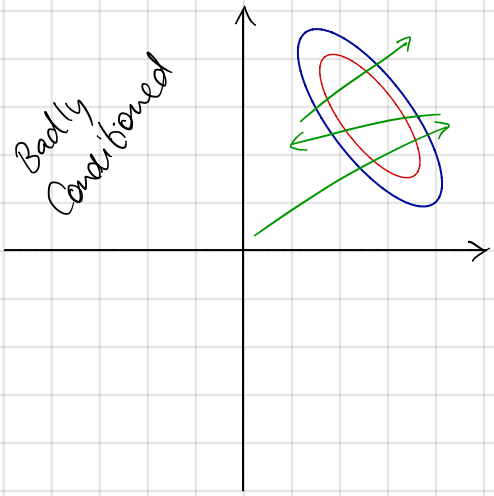
* Numerical Stability

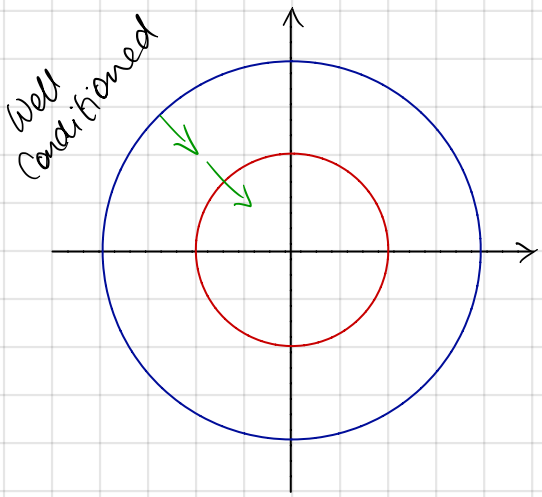
Big number + small numbers

→ Numerical errors (see ipython notebook)

Guiding principle - we want our variables to have mean zero and equal variance

i.e. $\mathbb{E}(x_i) = 0$ and $\sigma(x_i) = \sigma(x_j)$



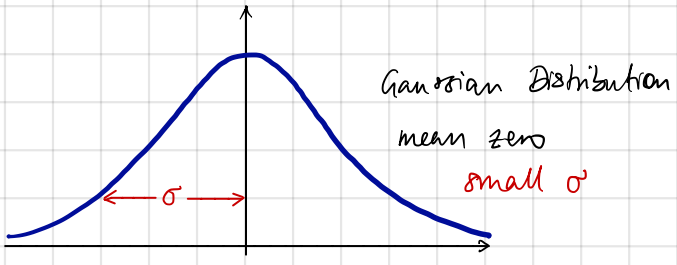


Images: $\frac{R-128}{128}$ $\frac{G-128}{128}$ $\frac{B-128}{128}$

* Weight initialisation

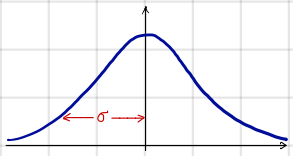
$D(S(Wx+b), L)$

W_0? b_0?



Because we combine the weights with the softmax function small σ results in more uniformly distributed probabilities and hence greater uncertainty while larger σ results in a more 'peaky' distribution and more certainty. To start we want the former as we want the optimisation to increase in certainty with time. Hence we choose small σ.

* Initialisation of the logistic classifier



$$\mathcal{L} = \frac{1}{N}\sum_i D(S(WX_i + b), L_i)$$

pixels − 128 / 128

ϕ

Optimisation

$$W \leftarrow W - \alpha \Delta_W \mathcal{L}$$
$$b \leftarrow b - \alpha \Delta_b \mathcal{L}$$

* Assignment 1 - not MNIST

See iPython Notebook

* Measuring performance

Split your data into three set

1. Training set
2. Validation set
3. Test set

Cross validation is very important in deep learning because there are lots of model parameters that can be tweaked.

* Validation and test set size

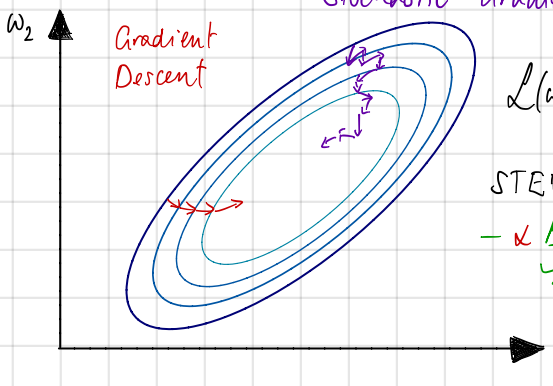
Rule of 30: A change that affects 30 samples in your validation set is usually statistically significant

N.B. do using the rule of 30, 30K samples in your validation set would enable you to observe one decimal place changes in accuracy i.e. 0.1%.

Here we assume your classes are balanced if not you'll need even more samples

* Optimising a logistic classifier - Stochastic Gradient Descent

Training logistic regression using gradient descent is nice in that you directly optimise for the error measure you're interested in but one issue is it doesn't scale very well



$\mathcal{L}(w_1, w_2) = \sum_i D_i$ ← All Data

STEP

$-\alpha \Delta \mathcal{L}(w_1, w_2)$ ← 3 × compute of $\mathcal{L}$

Derivative w.r.t weights

← iterative so we have to do this O(10) times

Stochastic gradient descent uses an estimate of the loss function $\mathcal{L}$, i.e. on a tiny random sample but instead iterates O(100) or O(1000) times rather than O(10) times.

The sample really must be random!

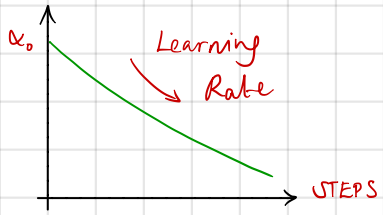
Helping SGD:

1. Inputs
   - Zero mean
   - Equal variance (small)
2. Initial weights
   - Random
   - Zero mean
   - Equal variance (small again)
3. Momentum
   - Take advantage of accumulated knowledge over previous steps
   - Use the running average of the gradients rather than the average of the current batch of data
   - Leads to better (smoother, faster) convergence

$\mathcal{L}(w_1, w_2) = \sum_i D_i$ RUNNING AVERAGE: $M \leftarrow 0.9M + \Delta\mathcal{L}$

STEP $-\alpha \Delta \mathcal{L}(w_1, w_2)$ $M(w_1, w_2)$

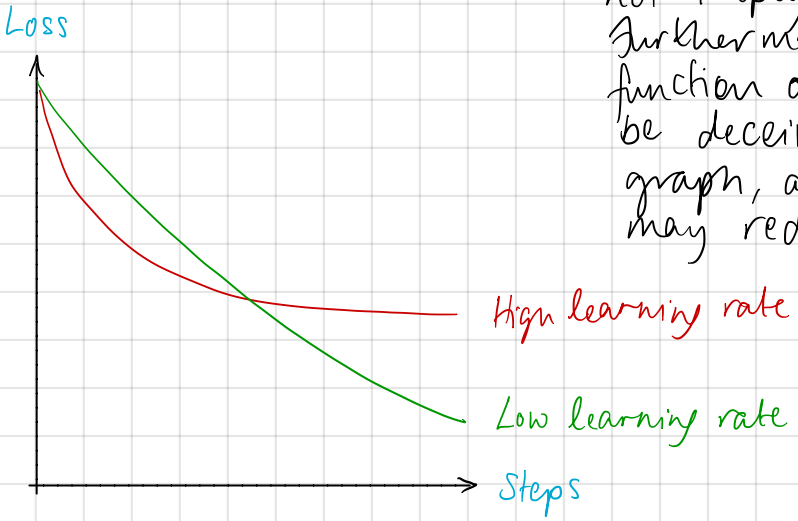
4. Learning rate decay - In utilising SGD over GD we take smaller, noisier steps
- Exponential decay is popular



STEP: $-\alpha \Delta(w_1, w_2)$
↑ Derivative or Momentum

* Parameter hyperspace

Learning rate tuning: A higher learning rate does not imply faster learning. Furthermore the loss as a function of the steps can also be deceiving. As shown in the graph, a high learning rate may reduce the loss quickly initially and plateau while a lower learning rate continues to decrease the loss.



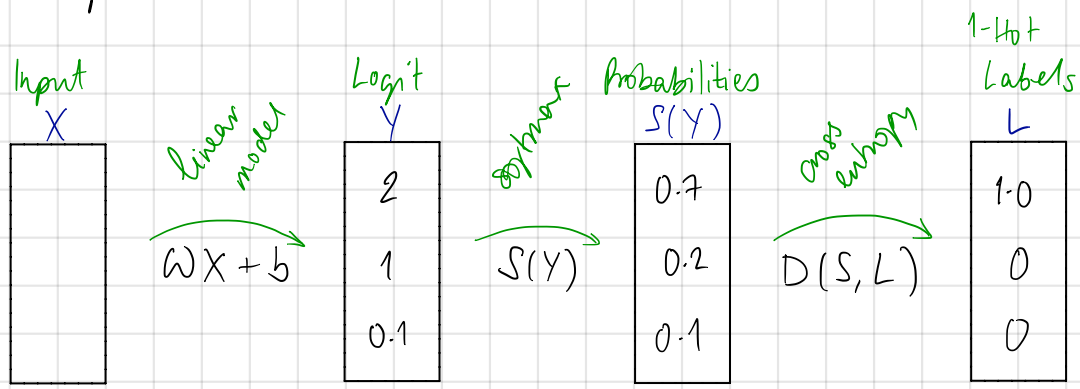
SGD -> 'Black Magic'

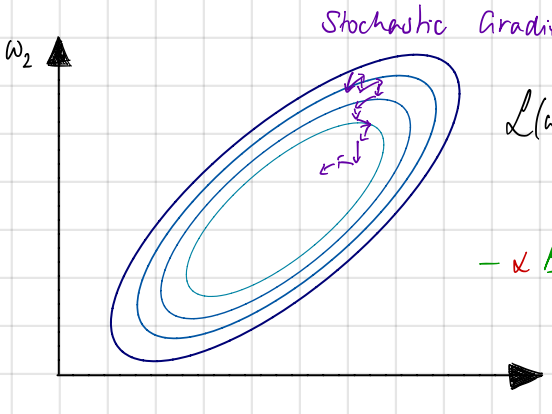
Many hyperparameters - initial learning rate
- learning rate decay
- momentum
- batch size
- weight initialisation

Remember: "Keep calm & lower your learning rate!"

ADAGRAD - does momentum and learning rate decay for you
- makes learning less sensitive to hyperparam's but performs a little worse than optimally tuned SGD with momentum

Recap





$$\mathcal{L}(\omega_1, \omega_2) = \sum_i D_i$$

$$-\alpha \Delta \mathcal{L}(\omega_1, \omega_2)$$

'SHALLOW' MODEL

INPUT → LINEAR → OUTPUT

NO DEEP LEARNING YET

L2 Deep Neural Networks

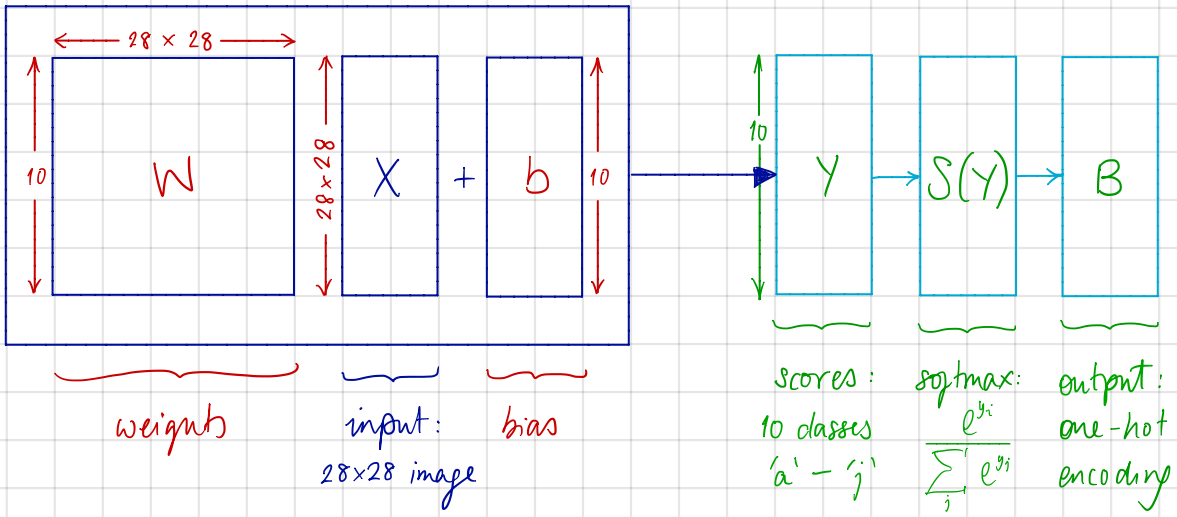
✻ Introduction to Lesson 2

→ In the previous lesson we trained a logistic classifier on images, now we'll turn this classifier into a deep network with just a few lines of code.

→ We'll look at how the optimiser works computing gradients for arbitrary functions

→ We'll look at regularisation which enables us to train much larger models

✻ Linear Model Complexity



Total number of parmeters
$= 28 \times 28 \times 10 + 10 = 7850$

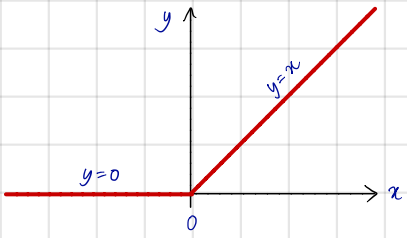
In fact in general for n inputs and k possible classes we have $(n+1)k$ parameters

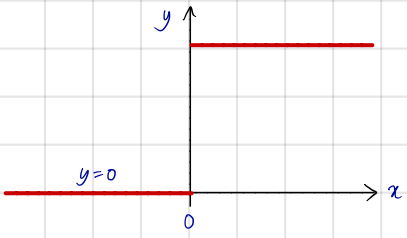
✻ Model limitations

Model is linear so can't represent multiplicative relations between inputs efficiently however

- Big matrix multiplication is exactly what GPUs were designed for, GPUs are cheap and fast
- Linear models are stable, small changes in inputs can only yield small changes in outputs
- Derivatives are constant – very nice and also stable

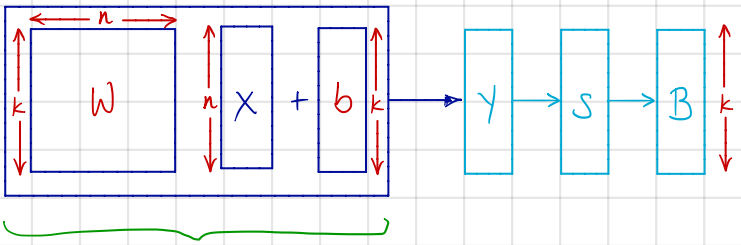
* Rectified Linear Units (ReLUs)



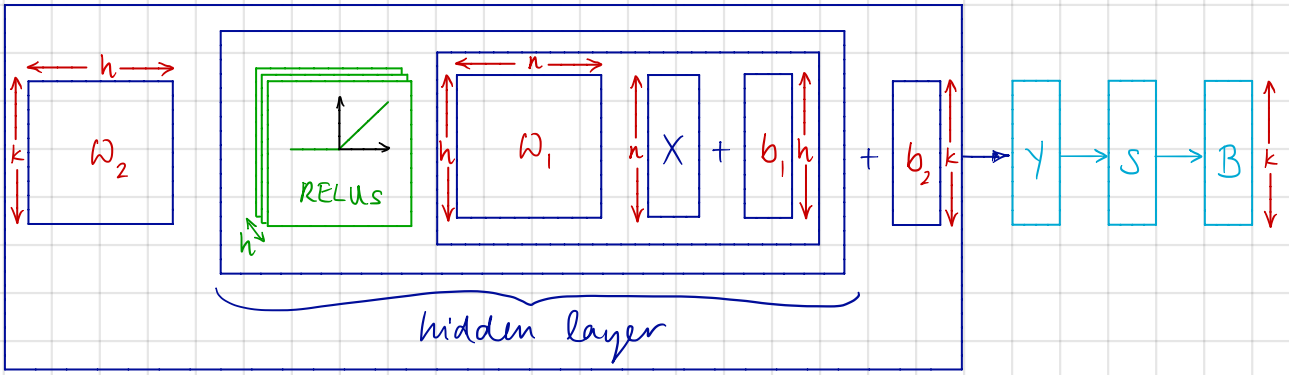


Before we had:

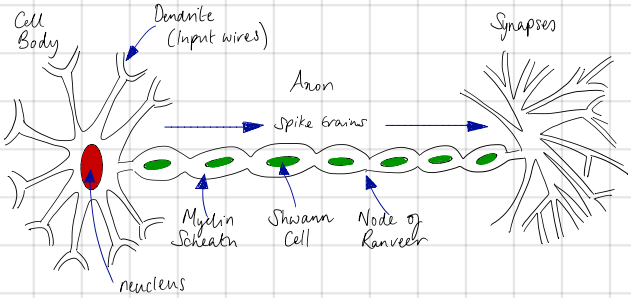
n = number of features
k = number of classes
h = number of hidden nodes

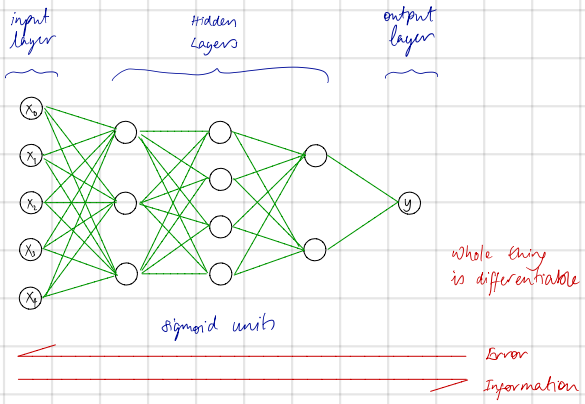


2 layer neural network:



* Typically when learning about neural networks comparisons are drawn to the brain and how it works. Diagrams of neurons and networks are drawn





We're not going to do that in this course.

This course is aimed at the lazy engineer with a big GPU who wants their machine learning to work better

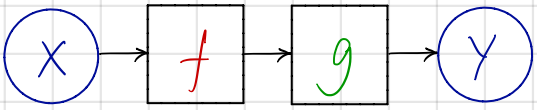
* Neural networks 'stack' up simple operations:

MULTIPLY → ADD → RELU → MULTIPLY → ADD → SOFTMAX

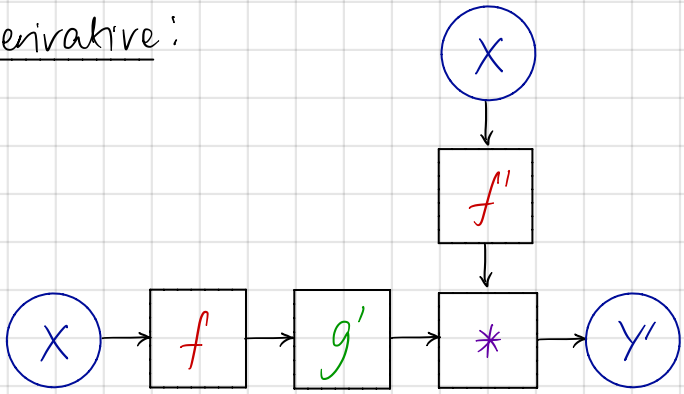
Chain R

$$[g(f(x))]' = g'(f(x))f'(x)$$

Function:

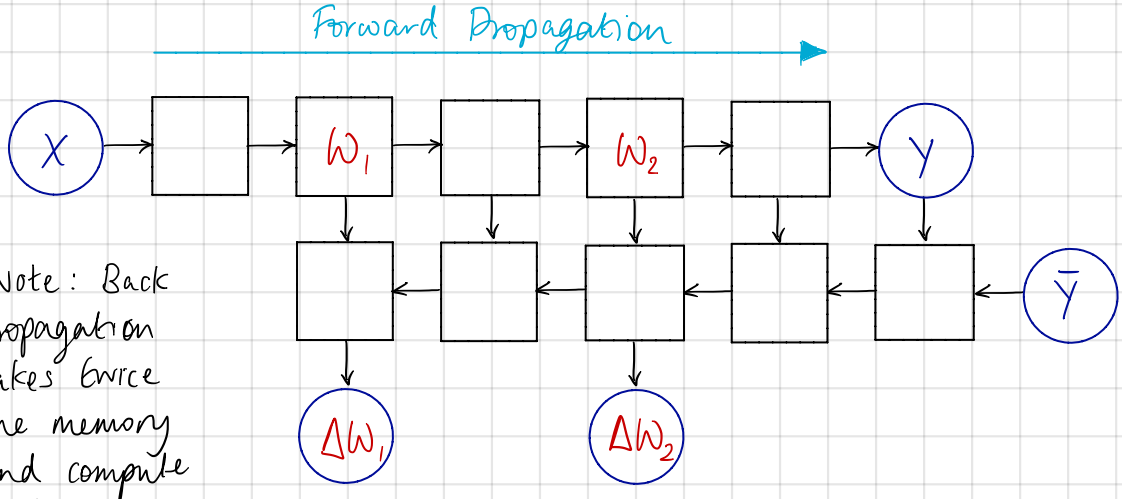


Derivative:



There is a way to write the chain rule which is extremely computationally efficient with lots of data reuse

* Back propagation



Note: Back propagation takes twice the memory and compute of forward propagation

$$W_1 \leftarrow W_1 - \alpha \Delta W_1 \qquad W_2 \leftarrow W_2 - \alpha \Delta W_2$$

* Assignment 2 - Stochastic Gradient Descent (SGD)

See iPython Notebook

Recap

1// Gradient Descent

Scores → Probabilities → Maximal Probability

Logits → Softmax → One Hot Encoding

compare using Cross Entropy

logits

cross entropy = $D(S, L) = -\sum_i L_i \log(S_i)$

one hot encoding

Loss = Average Cross Entropy (over all input datapoints)

$$\mathcal{L} = \frac{1}{N} \sum_i D(S(WX_i + b), L_i)$$

Minimise the loss to find the weights & biases using Gradient Descent

(loop) $W \leftarrow W - \alpha \Delta_W \mathcal{L}$

$b \leftarrow b - \alpha \Delta_b \mathcal{L}$

2// Stochastic Gradient Descent (Scales better)

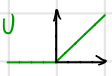
Calculating derivatives is expensive (3 times the cost of computing the loss). SGD calculates the loss and its derivatives on a random sample of the dataset. To improve stability we use the running average instead of just the average over that random batch.

3// Stochastic Gradient Descent with RELUs

2 layer (1 hidden) neural network:

$W_2 [R(W_1 X + b_1)] + b_2 \rightarrow Y \rightarrow S(Y) \rightarrow B$

RELU



Our data :
- Training set 200K
- Validation set 10K
- Test set 10K

image_size = 28 i.e. each input is a 28×28 matrix or when flattened a 784 array

num_labels = 10 (A, B, C, D, E, F, G, H, I, J)

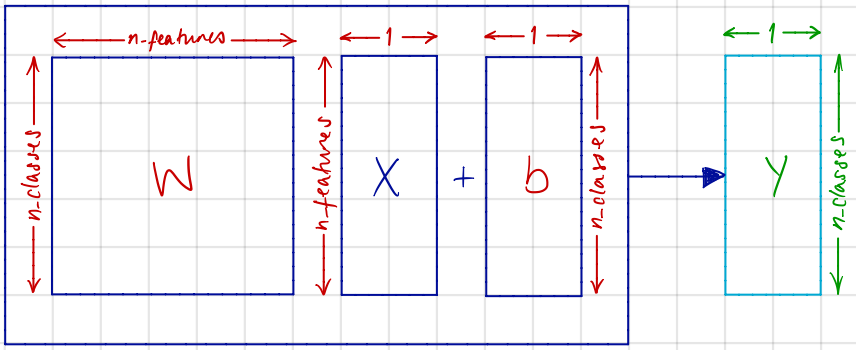
Gradient Descent : Training set = 10K
Nsteps = 801

Training on 200K examples would be slow

Stochastic Gradient Descent : Batch size = 128
Nsteps = 3001

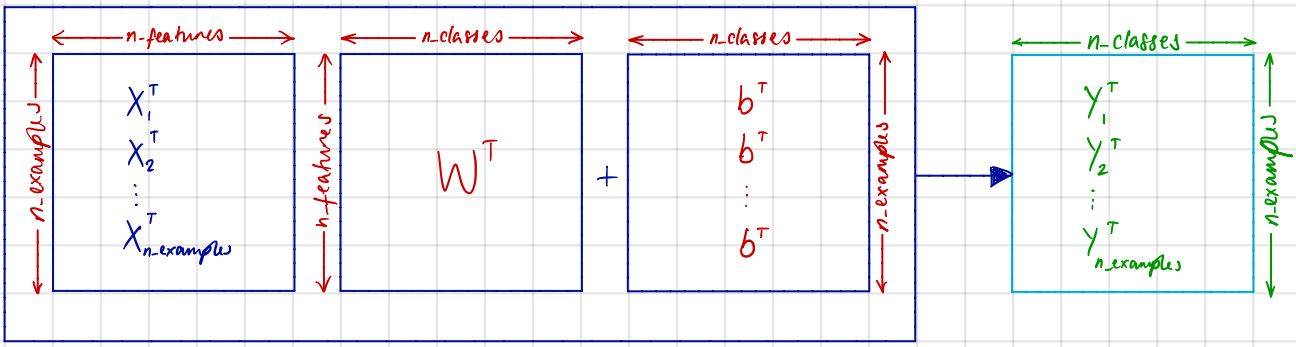
Some notes on implementation...

We show what happening inside the layer of a neural network like this



Here we input a single example X and obtain its output Y

But in the implementation we apply the linear transform to all the inputs at once so it really looks like this

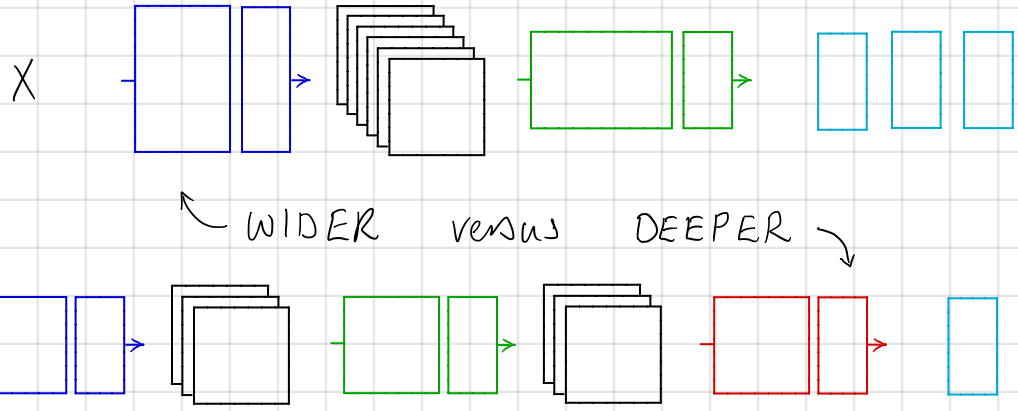


In tensorflow...

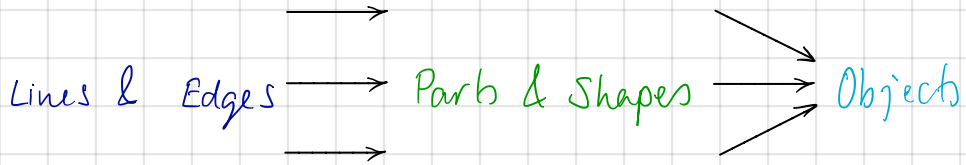
```
logits = tf.matmul(tf_train_dataset, weights) + biases
```

* Training a deep learning network

So in assignment 2 we made a small 2 layer neural network. We can increase its size/complexity by increasing the number of hidden nodes but it turns out this is not a particularly efficient way. We can do better by making the network deeper.



Many natural phenomena actually already have a hierarchical structure where more complex parts are made from simpler fundamental parts. Neural networks are well suited to these kinds of problems. The model structure matches the kind of abstractions you might expect to see in your data.



* Regularisation

Why have deep models not been found effective earlier?

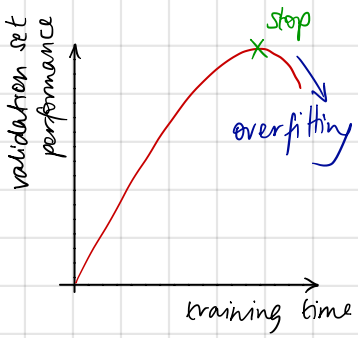
Deep models only really shine if you have enough data to train on and it's only in recent years that large enough datasets have made it into the academic world

Regularisation techniques for training very large models have improved substantially since neural networks were first introduced

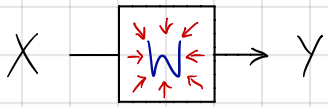
Skinny jeans problem:

Skinny jeans look great because they fit really well but are difficult to get into so most people wear jeans that are a little bit bigger. Similarly a network which is just the right size for your data is very difficult to optimise. Instead we train networks which are much bigger and then try our best to prevent overfitting.

Early termination:



Regularisation: Applying artificial constraints on the network that implicitly reduces the number of free parameters while not making it more difficult to optimise.

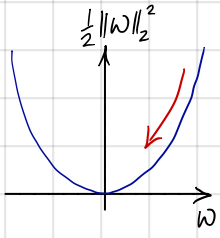


Think stretchy fabric skinny jeans!

L_2 Regularisation

$$\mathcal{L}' = \mathcal{L} + \beta \frac{1}{2} \|w\|_2^2$$

new loss | loss | small const. | weights

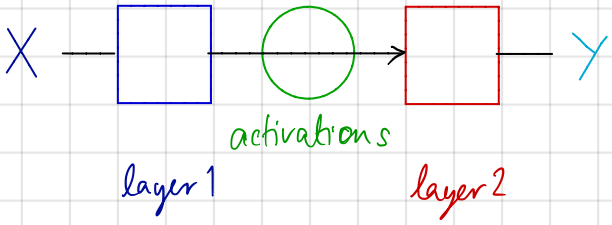


penalises large weights

↑ yet another hyperparameter to tune

✱ Dropout

Regularisation technique that has emerged very recently and works very well (Geoffrey Hinton)



For every example you train your network on, randomly set half your activations to zero and double the rest.

The effect of this process is to force your network not to rely on the information given by any one activation because it could be lost at any time. You end up with multiple activations propagating the same information. In other words you force your network to learn redundant representations. This sounds inefficient but actually makes the network robust. The network acts as if taking the consensus over an ensemble of networks, which is always a good way to improve performance

Dropout is one of the most important techniques to emerge in the last few years

If dropout doesn't work for you, you probably need a bigger network.

When you evaluate a network that's been trained using dropout you obviously no longer want this randomness, you want something deterministic. We want to take the consensus over the redundant models, this is found by averaging over the activations. Doubling the activations which are not zeroed during dropout means that during evaluation your activations are properly scaled and expectations are kept consistent. You simply use your activations (no zeroing half and doubling the rest) and average over them.

* Assignment 3: Regularisation of a deep network

See iPython Notebook

L3 Convolutional Neural Networks

* Introduction

If your data has some structure and your network doesn't have to learn that structure from scratch it's going to perform better.

For example in character recognition colour is not important so it is easier to learn the greyscale representation $((r+g+b)/3)$ rather than the colour (r, g, b) representation.

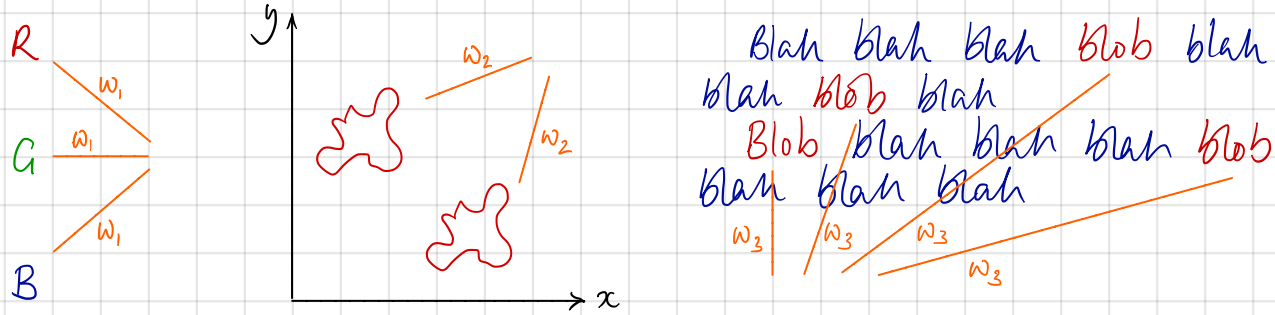
* Statistical Invariance

Example: Translation invariance - where the location of the image does not matter

Example: The meaning of a word is (mostly) independent of the location in the text

Weight sharing

When you know that two inputs can contain the same information then you share the weights and train the weights jointly for those inputs



IMAGES → CONVOLUTIONAL NEURAL NETWORKS

TEXT / SEQUENCES → EMBEDDING & RECURRENT NEURAL NETWORKS

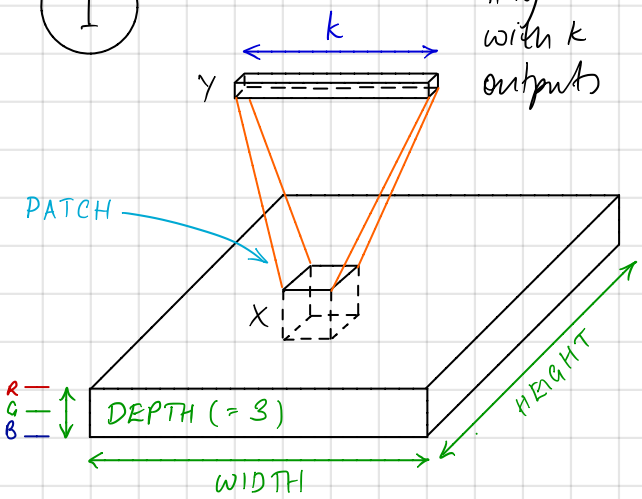
Statistical invariants: things which don't change on average across space or time are everywhere

✱ Convolutional Networks (Convnets)

Convnets are neural networks that share their parameters across space.

Example: Image

1

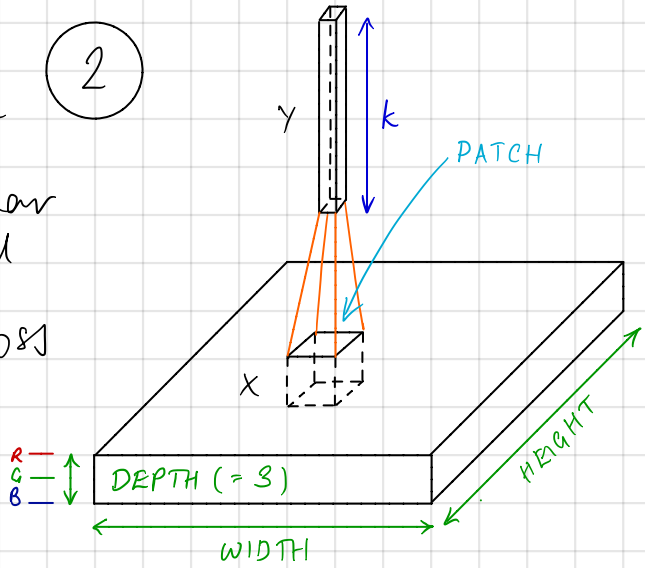


Tiny NN with k outputs

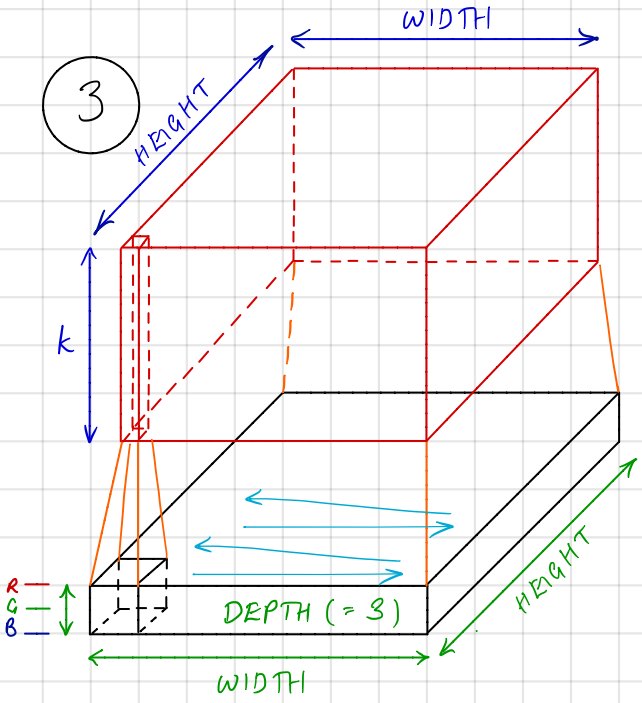
Lets represent our outputs vertically in a column as shown below and then slide the NN across the image without changing the weights systematically covering the whole image

If the patch was the same size as the original image then it would be no different to a regular layer of a NN but the small patch results in many fewer weights which are shared across space

2

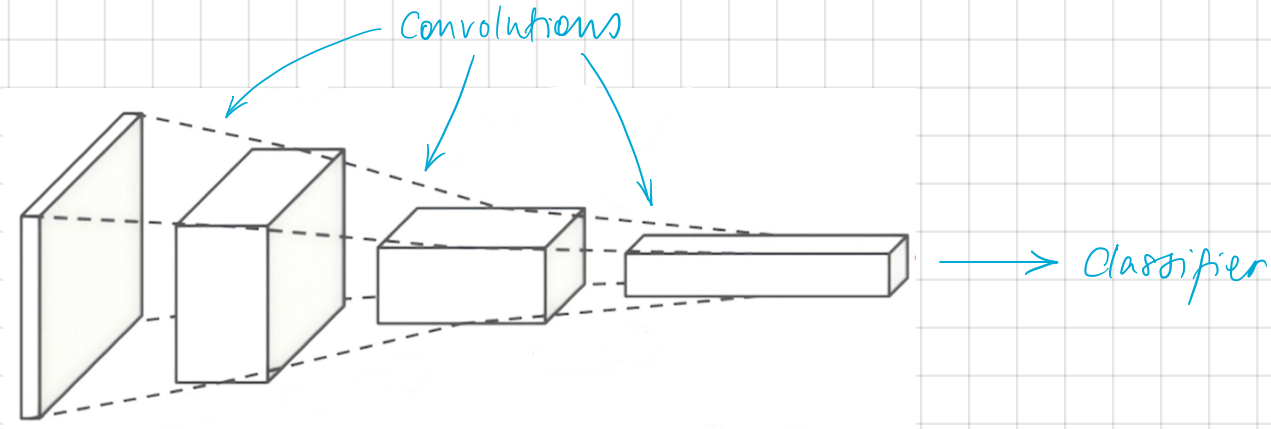


3



A convnet is a deep NN where instead of having stacks of matrix multiply layers we have stacks of convolutions

* Convolutional Pyramid

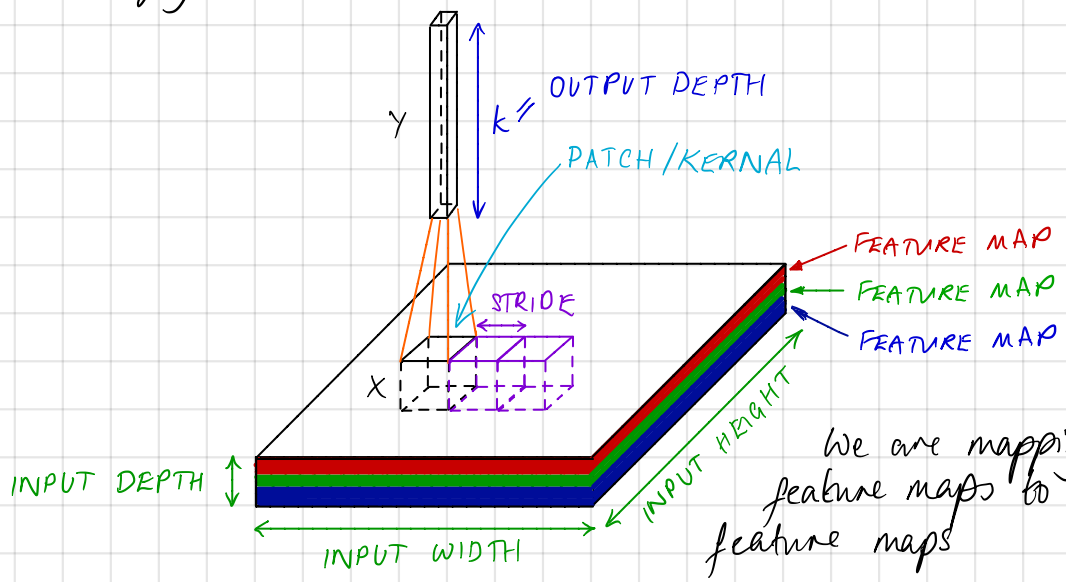


256 × 256 × 3 → 128 × 128 × 16 → 64 × 64 × 64 → 32 × 32 × 256 → SPACE DEPTH

We apply convolutions which progressively squeeze the spacial dimensions while increasing the depth which corresponds roughly to the semantic complexity of your representation

You end up with a representation where all the spacial information has been squeezed out and only parameters that map to the content of the image remain

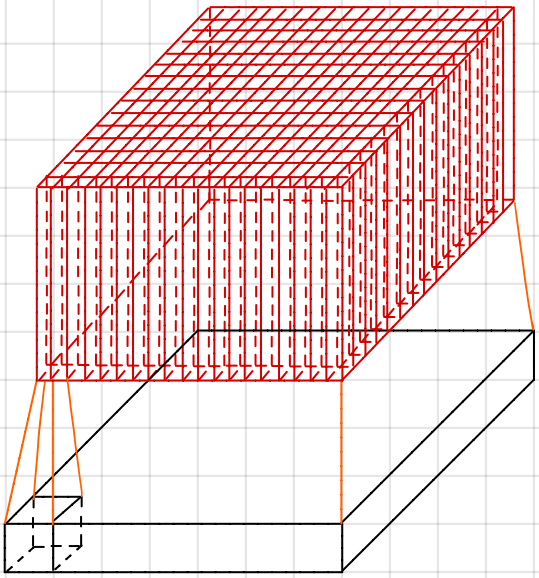
* Terminology



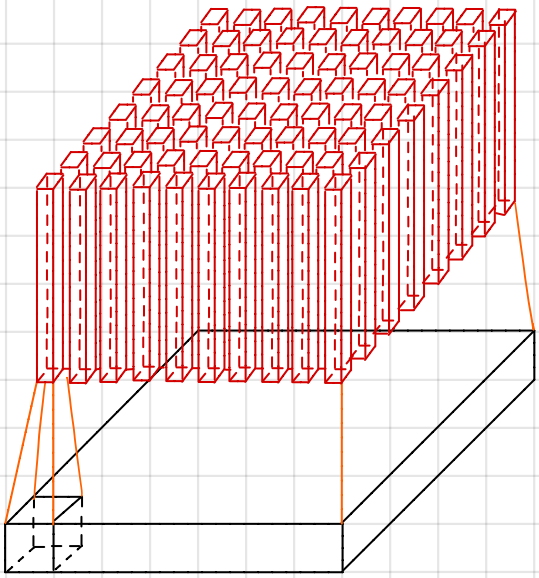
We are mapping 3 feature maps to k feature maps

The stride is the number of pixels you're shifting each time you move your filter.

A stride of 1 makes the output roughly the same size as the input.

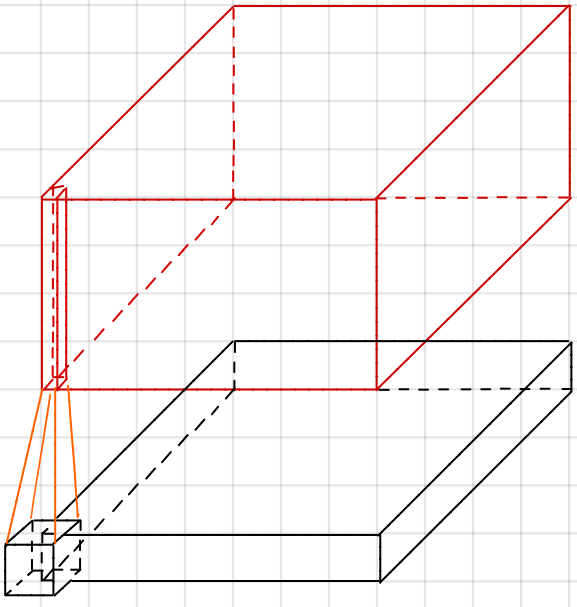


A stride of 2 makes the output dimensions roughly half those of the input.



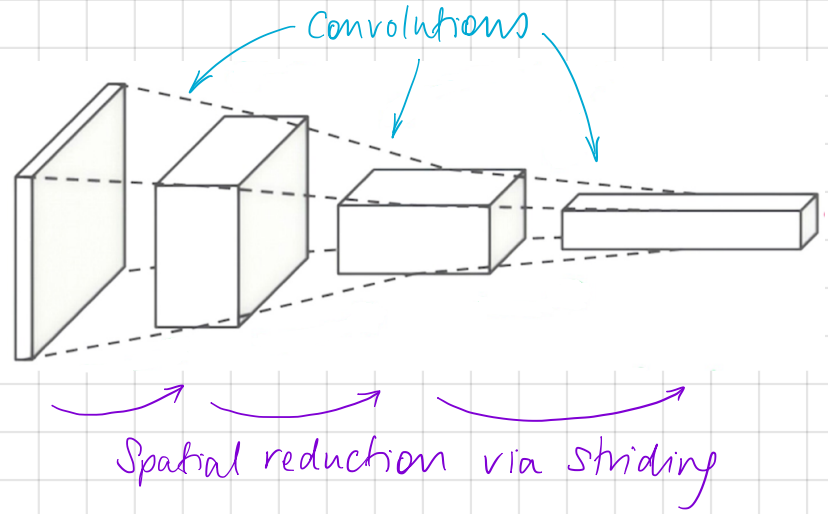
We say roughly because it depends what we do at the edges. Valid padding does not allow the patch to go over the edge of the image.

In same padding our patch goes over the edge of the image and we pad with zeros in such a way our output map size is exactly the same as the input map size.

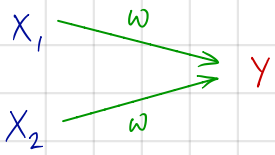


* Summary : building a simple convnet

1. Stack up your convolution pyramid
2. Use strides to reduce the dimensionality and increase the depth of your network layer after layer
3. Once you have a deep and narrow representation connect the whole thing to a few fully connected layers
4. Train your classifier!



* Chain rule with sharing



$$\frac{\Delta L}{\Delta w} = \frac{\Delta L}{\Delta w}(X_1) + \frac{\Delta L}{\Delta w}(X_2)$$

Add gradients for each patch

* Improvements on convnets

1. Pooling
2. 1 x 1 Convolutions
3. Inception architecture

Pooling

Striding can be an aggressive way to reduce the feature map size, it removes a lot of information. Instead of using a larger stride we can use a small stride but then take all the convolutions in a neighbourhood and combine them somehow. This idea is called pooling and there are a few ways of going about it

Max Pooling (Most common)

Take the maximum response in the neighbourhood

i.e.

$$Y = \max(X_i)$$

- Parameter free so doesn't increase the risk of overfitting
- Often more accurate
- Computationally more expensive (than taking larger strides)
- More hyperparameters
  - Pooling size
  - Pooling stride

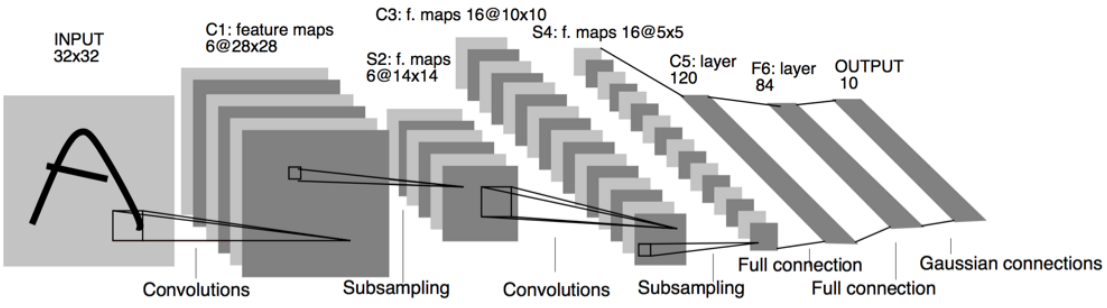
} don't have to be the same

Typical architecture for a convnet:

- Image
- Convolution
- Max Pooling
- Convolution
- Max Pooling
- Fully Connected
- Fully Connected
- Classifier

See 'LENET-5' Yann Lecun 1998
→ Character recognition (below)

and 'ALEXNET' Alex Krizhevsky 2012
→ ImageNet object recognition challenge winner



Average Pooling

Take the average response in the neighbourhood

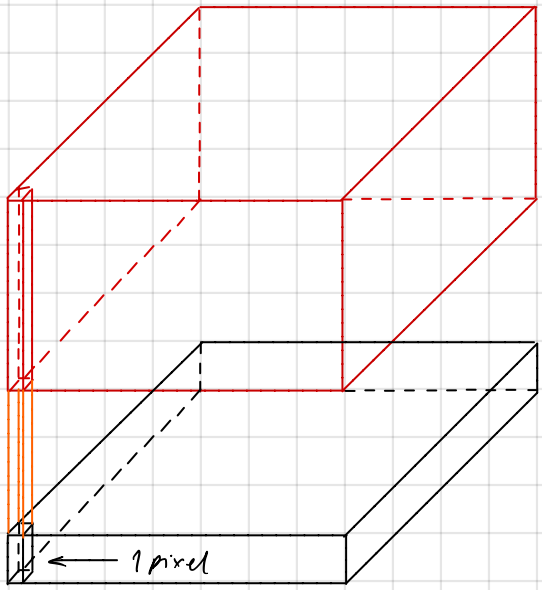
i.e.

$$Y = \text{mean}(X_i)$$

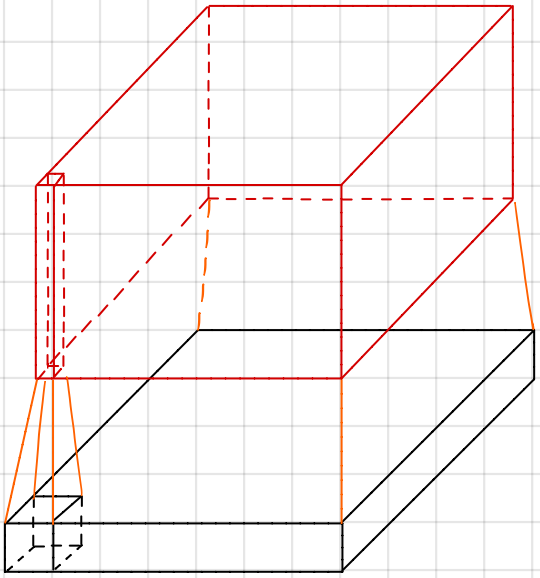
It's a bit like lowering the resolution of the image

* One-by-one Convolutions

One-by-one convolution

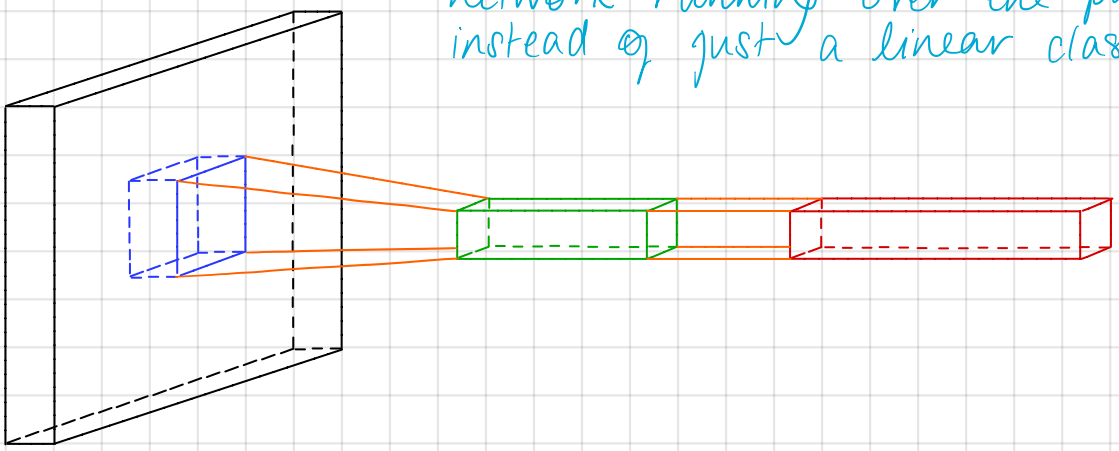


Regular convolution



Small linear classifier over a patch

Adding a one-by-one convolution in the middle → mini neutral network running over the patch instead of just a linear classifier



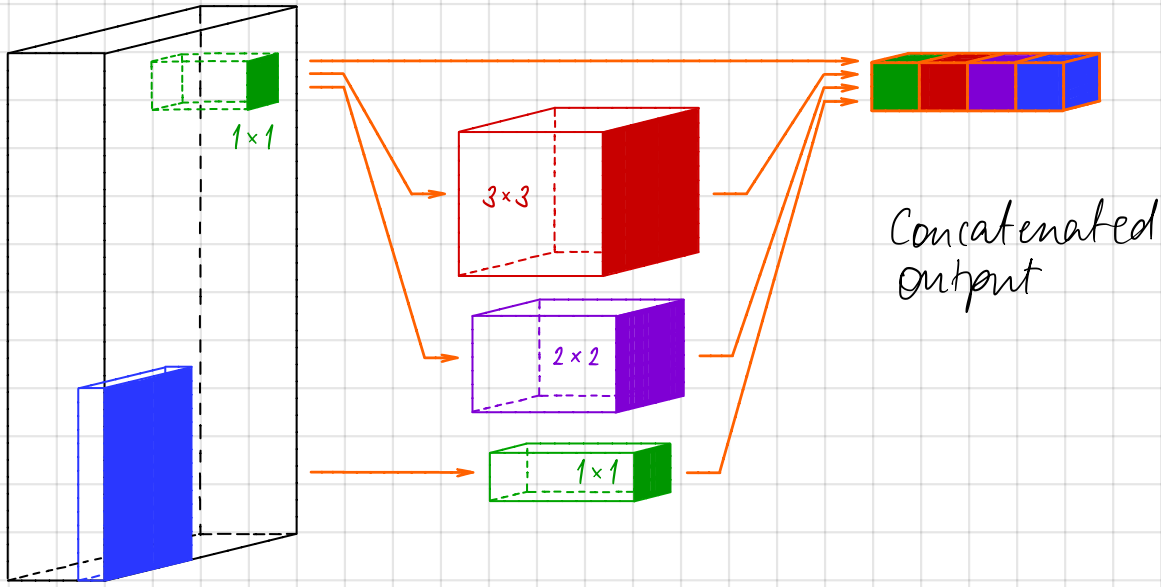
Interspersing convolutions with one-by-one convolutions is an inexpensive way to make models deeper and have more parameters without completely changing their structure. They are cheap because they reduce to matrix multiplications.

Pooling and one-by-one convnets lead us to a general strategy which has been found to be very successful at creating convnets that are both smaller and better than convnets that simply use a pyramid of convnets.

* Inception Modules

At each layer of your convnet you can make a choice about its type. You could have a pooling operation or a convolution. For the latter you would need to decide the patch size. Is it a 3×3 convolution or a 2×2 or a one-by-one? All of these are beneficial to the modelling power of your network, so why choose? Let's use them all.

Instead of having a single convolution we have a composition of them, i.e. an inception module:



What's interesting is that we can choose the parameters in such a way that the total number of parameters in the model is very small and yet performs better than a simple convolution

* Conclusion

Neural networks provide a general framework with lots of small building blocks. You can explore ideas quickly and come up with interesting model architectures for your problem

See V. Domoulin & F. Visin, A guide to convolution arithmetic for deep learning

Convnets is where GPUs begin to shine because they are computationally expensive.

* Assignment 4 - Convolutional Models

See iPython Notebook

L4 Deep Models for Text & Sequences

* Train a Text Embedding Model

Take for example the problem of document classification based on the words in the document. This is hard because ...

1. There are lot of words
2. Most words are rarely seen

In fact rare words tend to be the most important e.g. the word 'the' tells us nothing about the document in which it is found but if we see the word 'retinopathy' it is more than likely a medical document. The word 'retinopathy appears with a frequency of 0.00001 % in English.

In deep learning rare events are a problem, we need a very large training set

* Semantic Ambiguity

CAT and KITTY are very different words but with similar meanings.

Recall that it helps to share parameters between similar things but to do this with the words CAT and KITTY, we're going to have to learn that they are related.

1. We want to see important words often enough to be able to learn their meaning
2. We also want to learn how words relate to each other so we can share parameters between similar words

To do this for any task that matters would require a huge amount of labelled data, which simply doesn't exist.

* Unsupervised Learning

Finding unlabelled text to learn from is easy, its everywhere.

* Embeddings

Idea: Similar words tend to occur in similar contexts

The words CAT and KITTY can be used in the same context e.g.

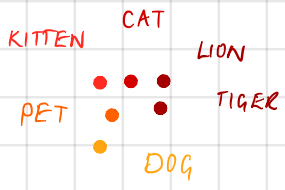
The CAT / KITTY purrs
This CAT / KITTY hunts mice

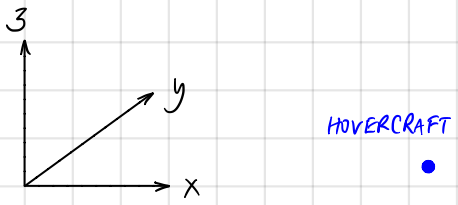
So let's instead try to predict a words context. A model that is good at predicting a words context will output similar values for CAT / KITTY

In this approach we don't need to know the words exact meaning, instead it is inferred from the surrounding words.

We use the idea that similar words occur in similar contexts to map words to small vectors (or embeddings) which are going to be close to each other for words which have similar meanings and far apart for words which don't

Embedding solves some of the sparcity problem.





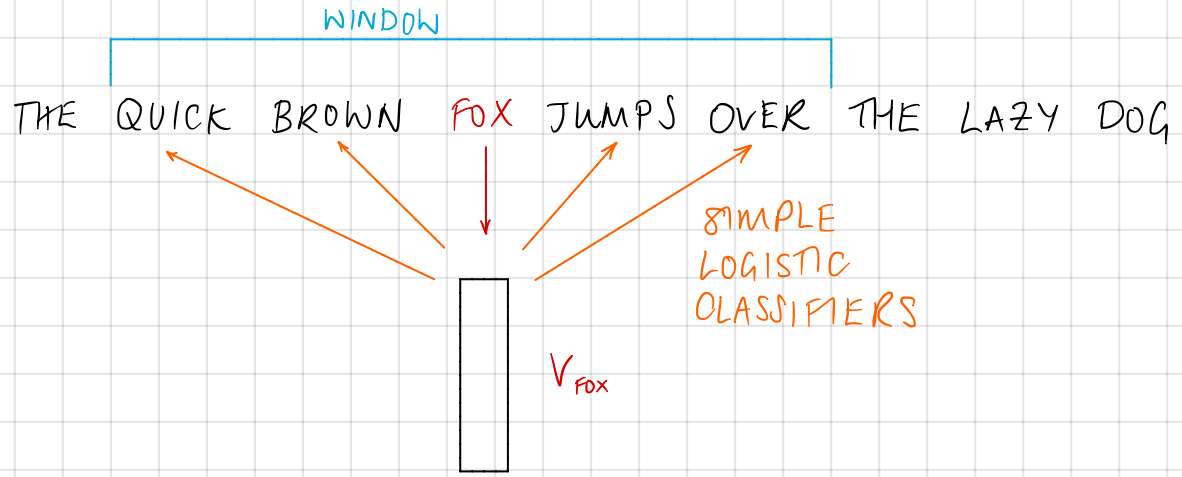
Once we have our embeddings our model no longer has to learn new things for every way there is to talk about a cat, it can generalize from the patterns seen in cat-like things

→ SHARING
→ GENERALISATION

* Word2Vec

Simple model to obtain embeddings that works very well.

Suppose we have the following sentence in a corpus of text,



We map each word in the sentence to an embedding, initially a random one. We then use the embedding to try and predict the context of the word.

In this model, the context is simply the neighbouring words. We pick a random word in a window around our original word and make that our target. Then train the model as if it were a supervised problem. To this we use a simple logistic regression (not a deep network)

* tSNE

We need to be able to check that our embeddings cluster as we would expect. There are a few ways to do this

1. Nearest neighbour lookup
2. Dimensionality reduction of the embedding space to two dimensions

for 2. PCA doesn't work, we need a method which preserves distances → tSNE

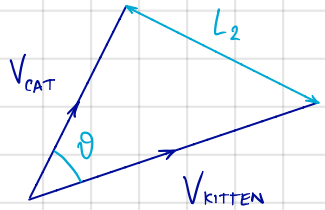
* Word 2Vec details

1. Measuring proximity:

Because of the way embeddings are trained, it's often better to measure proximity using a cosine distance instead of L2 for example

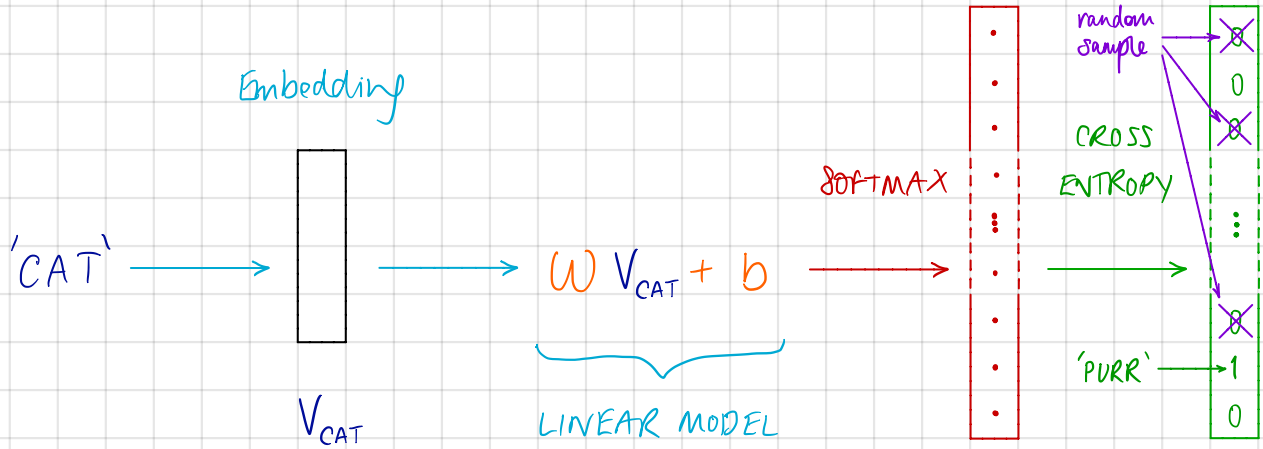
L_2 : $\| V_{CAT} - V_{KITTEN} \|_2^2$

Cosine θ : $\frac{V_{CAT} \cdot V_{KITTEN}}{\| V_{CAT} \| \| V_{KITTEN} \|}$



In fact it's often better to normalise all embedding vectors

2. Predicting from a very large set of words:



The problem is we have many many words in our vocabulary an computing the softmax function over all the words can be very inefficient. So we can use a trick...

Sampled Softmax

Instead of treating the softmax as if the label has a probability of one and all the other words have a probability of zero, we sample the words that are not the target, pick only a handful of them and ignore the others. This technique of sampling the negative targets for each example is called Sampled Softmax and it makes things faster at no cost in performance.

* Word Analogies

Semantic analogy : Puppy → Dog / Kitten → Cat

$$V_{PUPPY} - V_{DOG} \sim V_{KITTEN} - V_{CAT}$$

Syntactic analogy : Taller → Tall / Shorter → Short

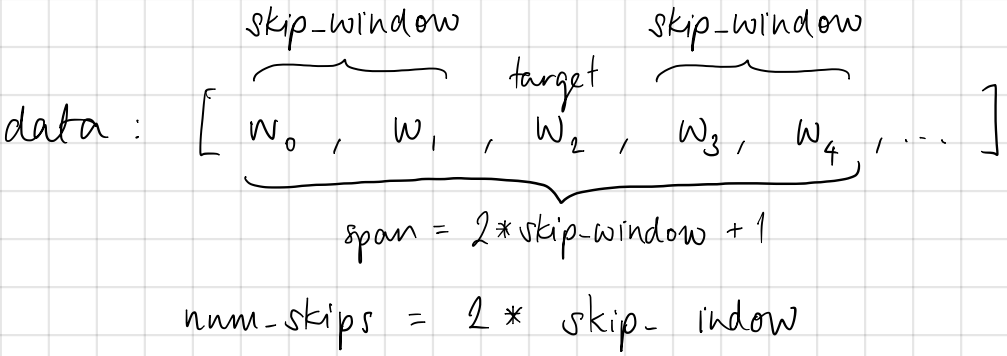
$$V_{TALLER} - V_{TALL} \sim V_{SHORTER} - V_{SHORT}$$

So for a good model one should find that in embedding space, $V_{PUPPY} - V_{DOG} + V_{CAT}$ close to V_{KITTEN}.

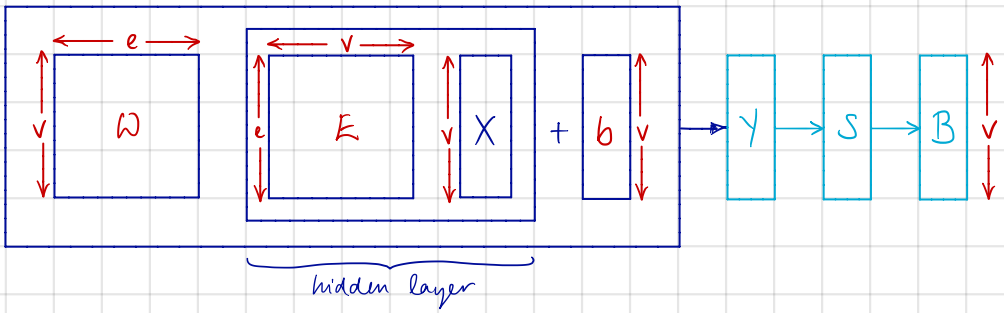
* Assignment 5 : Word2Vec and CBOW

See iPython Notebook

Word 2 Vec Model Notes :



num_skips = 2 * skip_window

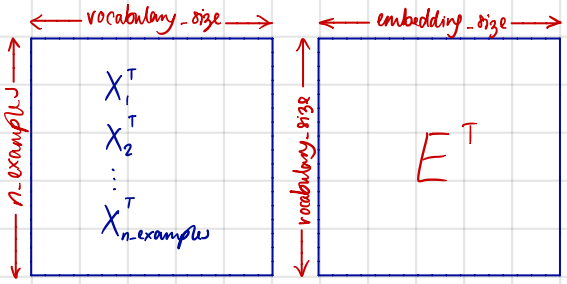


The hidden layer - weights matrix is the embedding matrix
- has no biases and no activation functions
- outputs the embedding vector for X (size e)

X and B are one-hot-encoded words from our vocabulary (size v)

Some notes on the implementation ...

Recall that tensorflow does the linear transform on more than one example at a time so in our hidden layer we really have



The code in tensorflow isn't written in a way that makes it obvious that we have a two layer (one hidden) neural network. It is written in a way that treats the embedding matrix as a transformation on the inputs to obtain the embedding vectors. These embedding vectors are then treated as new inputs to a classifier.

Tensorflow also has some infrastructure specifically for dealing with embeddings. We don't just do the usual matrix multiplication. Instead of feeding in the one-hot-encoding for each example, we feed the index corresponding to the element which has value one (all others have value zero) and look-up its embedding in the embedding matrix.

So in the code

train_dataset has shape [batch_size]

embeddings has shape [vocabulary_size, embedding_size]

and to get our embedding vectors we do

embed = tf.nn.embedding_lookup(embeddings, train_dataset)

* Skip-gram model

word → Target_word Embedding → Context_word

e.g. (num_skips, skip_window) = (2, 1)

batch = $[w_1, w_1, w_2, w_2, \dots]$

embeddings = $[e_1, e_1, e_2, e_2, \dots]$

labels = $[w_0, w_2, w_1, w_3, \dots]$

* Continuous Bag of Words Model

Context_words → Sum of all context_word embeddings → Target_word

e.g. (num_skips, skip_window) = (2, 1)

batch = $[(w_0, w_2), (w_1, w_3), \dots]$

embeddings = $[(e_0 + e_2), (e_1 + e_3), \dots]$

labels = $[w_1, w_2, \dots]$

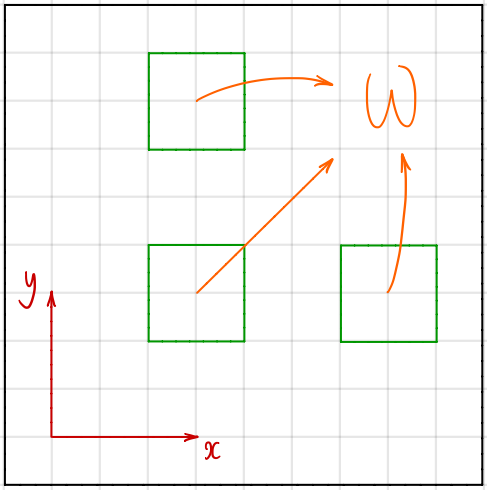
* Sequences of varying length

So far our models have all only dealt with inputs of a fixed size, e.g. in Word2Vec we have one word as an input which we turn into a vector of fixed size.

When dealing with data like speech or text we no longer have fixed size inputs

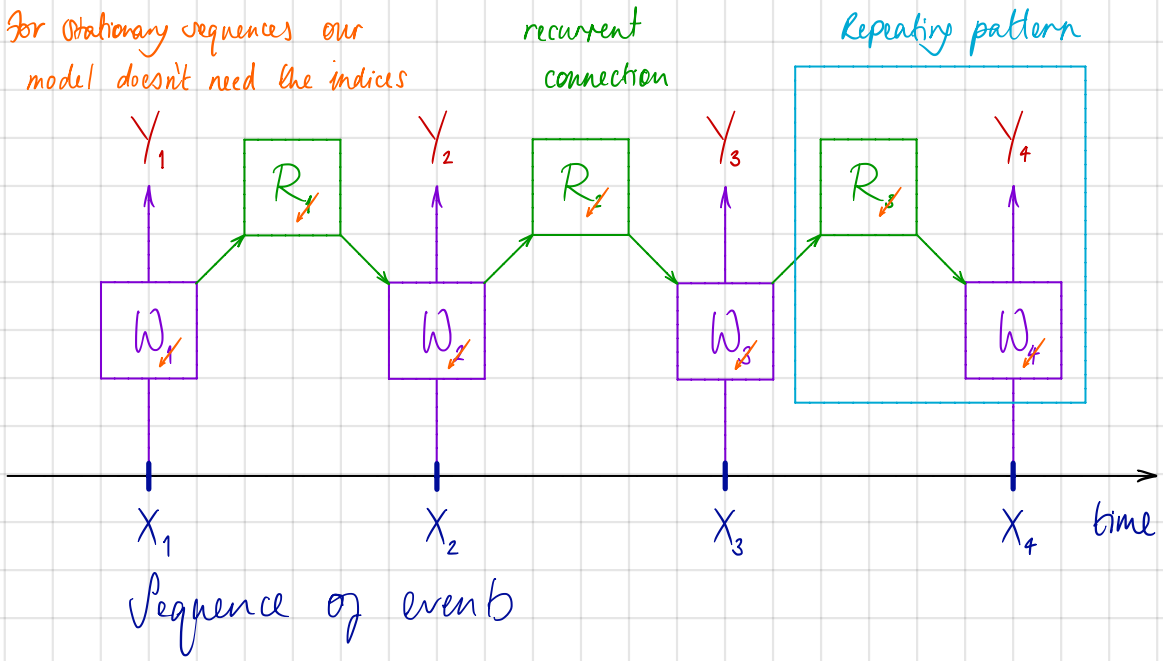
* Recurrent Neural Networks

Convolutional Networks:



Recurrent Networks:

In convolutional networks we share parameters across space to extract patterns over an image. In recurrent networks we do the same but over time instead of space.



Sequence of events

At each point in time we want to make a decision about what's happened so far in the sequence.

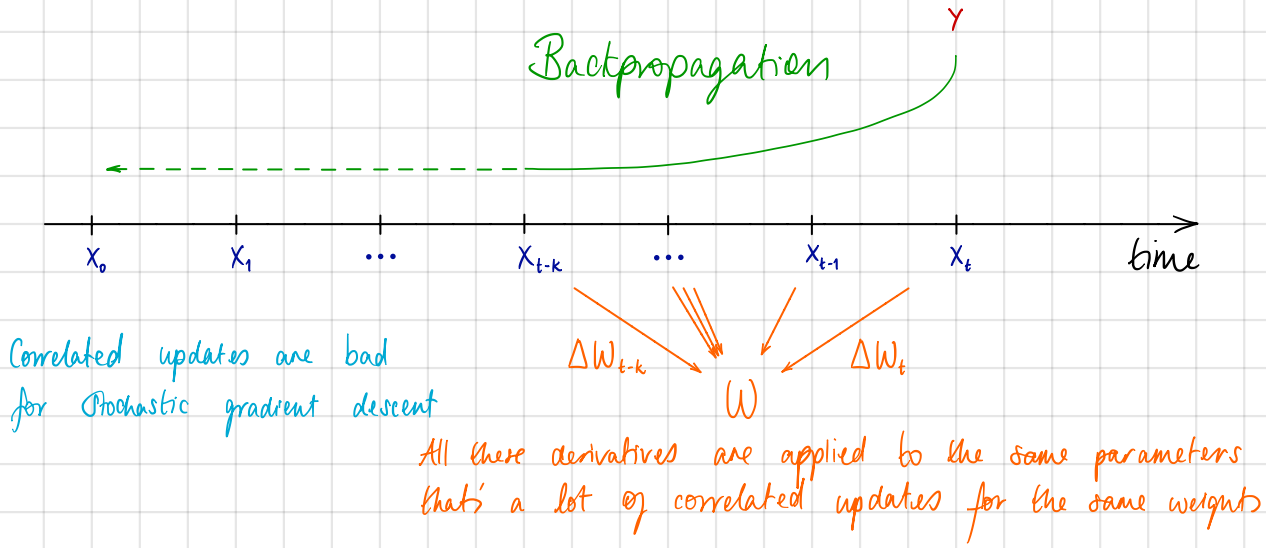
For sequences which are reasonably stationary we can use the same classifier at each point in time, i.e. $W_1 = W_2 = W_3 = \dots$

Since it is a sequence we also want to take into account the history. To do this we can use the state of the previous classifier, as a summary of past events, recursively.

The further in the past one wants to remember, the deeper the neural network would need to be. You would need a layer for each event in your sequence. That could be hundreds or thousands of layers. Instead of doing this we use tying and have a single model responsible for summarising the past and providing that information to the classifier.

This construction gives us a network with a relatively simple repeating pattern with part of the classifier connecting to the input at each time step and another part called the recurrent connection connecting the classifier to the past at each step

* Backpropogation through time



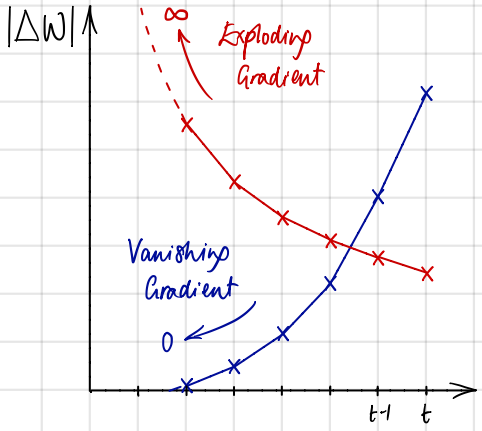
Correlated updates are bad for Stochastic gradient descent

All these derivatives are applied to the same parameters that's a lot of correlated updates for the same weights

To compute the parameter updates of a recurrent network we need to backpropagate the derivative through time to the beginning of the sequence (or in practice as many steps as we can afford)

Stochastic gradient descent prefers to have uncorrelated updates to its parameters. Correlated updates make the training very unstable...

* Exploding & Vanishing Gradients



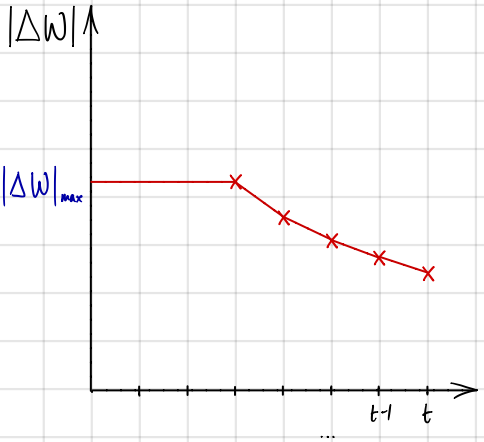
Either the gradients grow exponentially
- they EXPLODE

Or the gradients decay very quickly
- they VANISH
- no training

We're going to fix these two problems very differently

Exploding gradients: Gradient clipping (simple hack)

$$\Delta W \leftarrow \Delta W \frac{|\Delta W|_{max}}{max(|\Delta W|, |\Delta W|_{max})}$$



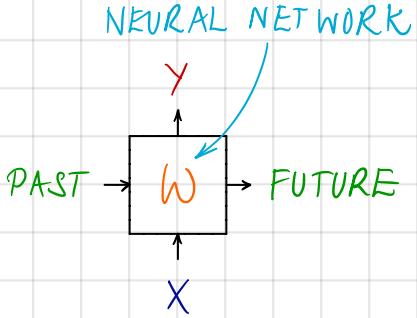
In order to stop the gradients growing unbounded one computes the norm and shrink the step when the norm grows too big

HACKY - CHEAP - EFFECTIVE!

Gradient vanishing: This problem is harder to solve. Vanishing gradients make your model's memory short which means a recurrent neural network tends not to work past a few time steps. MEMORY LOSS IN RNNS
This is where LSTMs come in.

* Long Short-Term Memory (LSTM)

Conceptually a recurrent neural network consists of a repetition of simple little units like this. We replace the NN with an LSTM cell



* Memory Cell

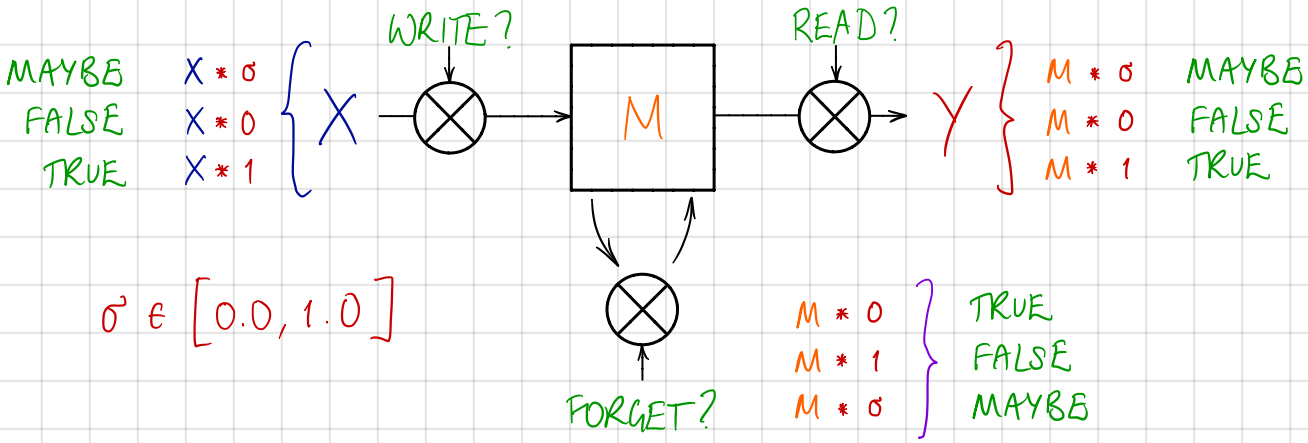
The aim here is for RNNs memorise things better.

What do LSTM cells look like?

A simple model of memory

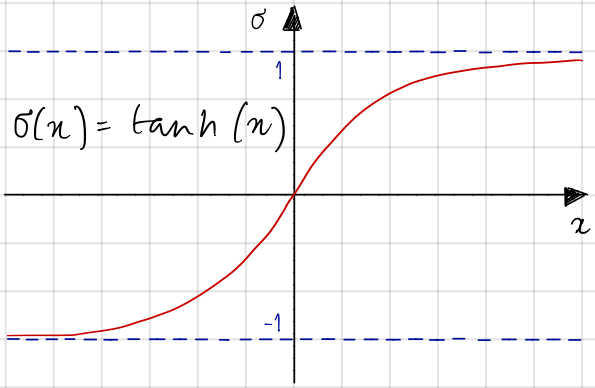
Instruction	Input	Output
1. WRITE	X	M
2. READ	M	Y
3. FORGET	M	

We can also draw this in a diagram



* LSTM Cell

Now suppose, instead of binary decisions we have continuous ones, i.e. we make our multiplicative factor $\sigma \in [0,1]$ continuous and differentiable.



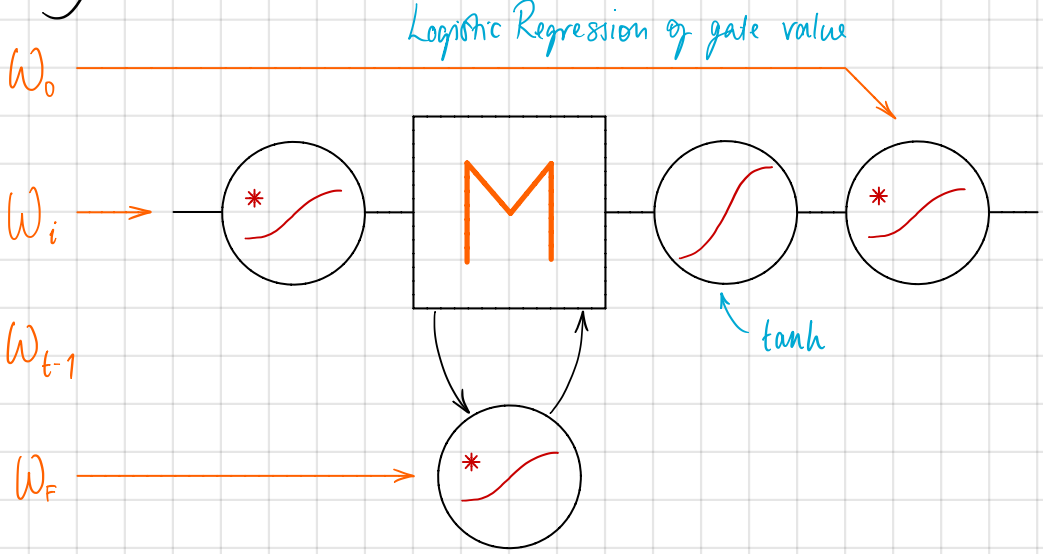
This gives us an LSTM.

We can take derivatives and backpropagate through it.

The gating values in each case are controlled by a tiny logistic regression on the input parameters. Each of them has its own shared parameters. To keep the outputs between one and minus one there's an additional $\tanh(x)$ at the output gate.

It looks complicated, but once you write down the formulae it's literally five lines of code.

It's well behaved, continuous and differentiable all the way which makes optimising the parameters very easy



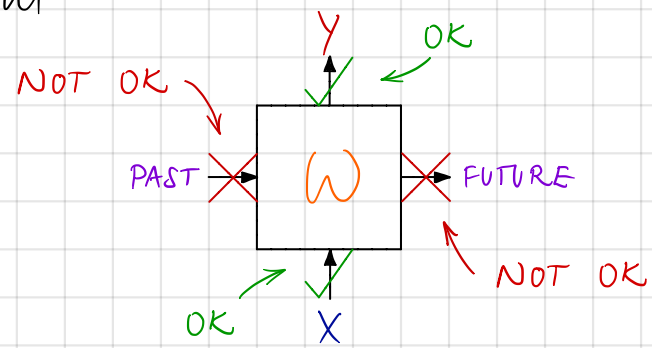
Why do LSTMs work?

In short, the gates help the model retain memories longer when it needs to and ignore events when it should. This makes the optimisation easier and solves the problem of gradient vanishing.

* LTSM Regularisation

One can always use L_2 Regularisation. Dropout also works well as long as it is used on the inputs and outputs and not on the recurrent connections

Dropout:



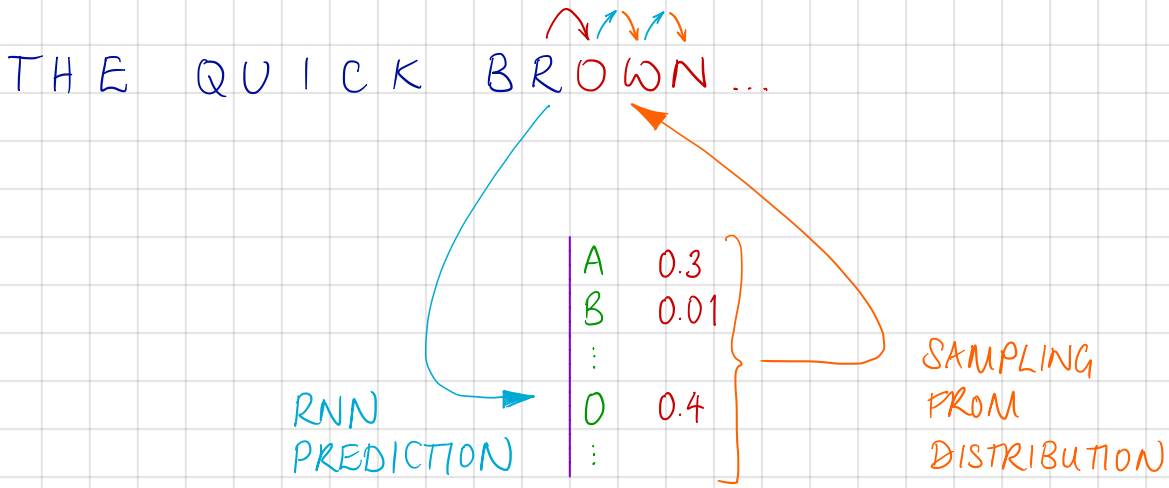
* Beam Search

What can you do with an RNN?

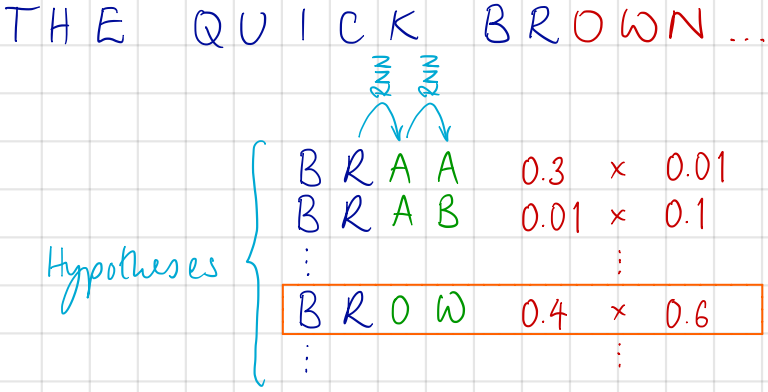
Lots of things but here is one.

Generating sequences

A model that predicts the next step in a sequence can be used to generate sequences, for example text, either one word or one character at a time. To do this one takes the sequence at time t, generates a prediction and then samples from the predicted distribution, i.e. pick one element, feed that sample to the next step and repeat, sample, predict, sample, predict, ...



This method is very greedy. An alternative would be to sample multiple times. We end up with multiple sequences (hypotheses). We then choose the best sequence based on the total probability of all the characters generated. This method avoids the problem of being stuck with one bad decision forever.



Problem:
Number of sequences we need to consider grows exponentially

Solution:
Beam search

In Beam search we keep only the most likely few candidate sequences at each time step and prune the rest. This works very well in practice

* Assignment 6: LSTMs

See ipython Notebook

Problem One:

LSTM cell dep contains four matrix multiplications that look like:

$$I \otimes X + O \otimes M + b$$

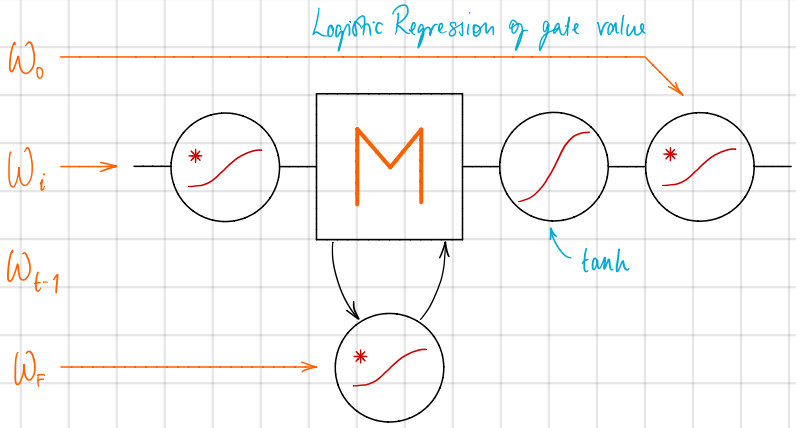
X = vocabulary_size × num_nodes

M = num_nodes × num_nodes

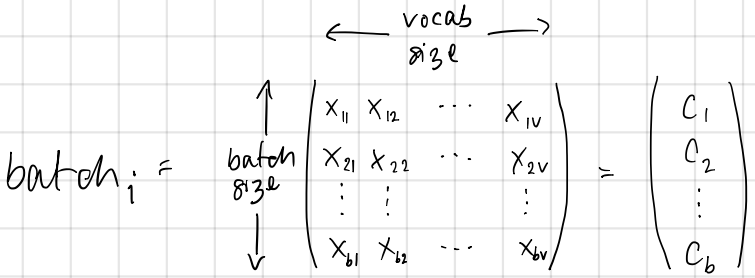
b = batch_size × num_nodes

$\Rightarrow$ I = batch_size × vocabulary_size

$\Rightarrow$ O = batch_size × num_nodes



Problem Two:



each row of batch i is the one-hot-encoding for the ith character in a series of batch-size strings

batch_size = 64 vocabulary_size = 27

Example:

text_size = 10 batch_size = 2 segment = 10/2 = 5

cursor = (0, 5) → (1, 6) → ... each word containing num_unrollings chars

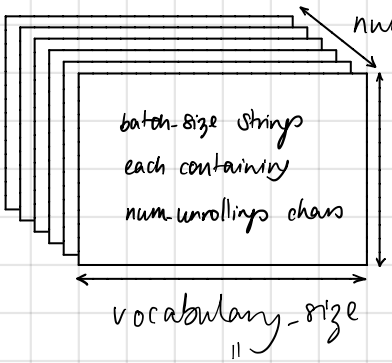
text = (- - - - - - - - - - - - - - - -

batch 1 c_1^1 ← segment → c_2^1 ... c_b^1 → batch 1

batch 2 c_1^2 ... c_2^2 ... c_b^2 → batch 2

batch 3 c_1^3 c_2^3 ... c_b^3 → batch 3

num_unrollings = 3



num_unrollings + 1 = 11

batch_size = 64

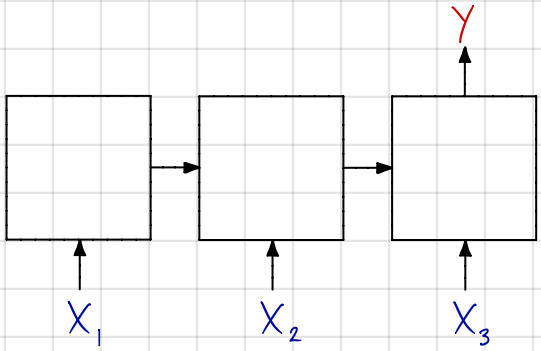
= { ith row of the jth batch
one-hot-encoding for the
jth char of ith string

* Lego

We can build deep model architectures from lego like components and then use backpropagation to optimise end to end

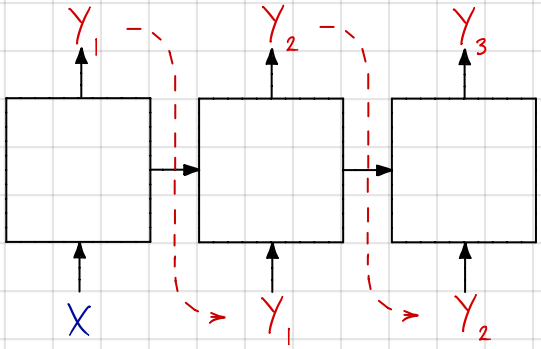
* Captioning and Translation

Variable length sequence in $X_1, X_2, X_3 \rightarrow Y$ Fixed length vector out

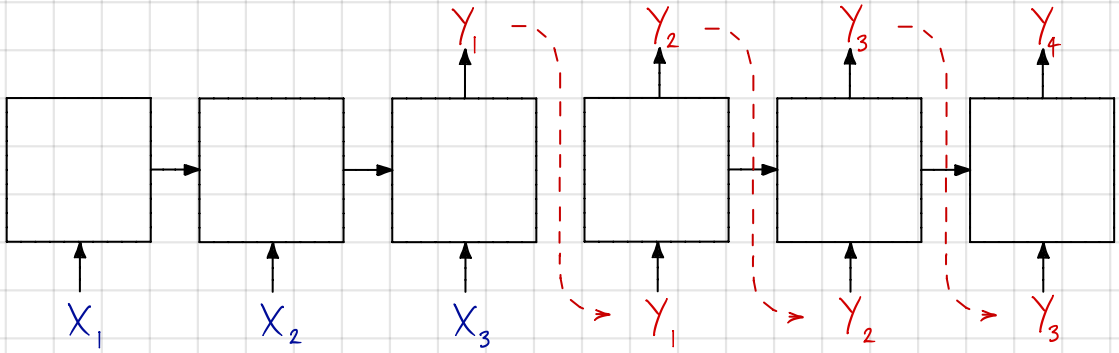


Sequence generation and beam search means we can also do the opp.

Fixed length vector in $X \rightarrow Y_1, Y_2, Y_3$ Variable length sequence out



Putting the two together ... $X_1, X_2, X_3 \rightarrow Y_1, Y_2, Y_3, Y_4$



$X_1, X_2, X_3 \rightarrow Y_1, Y_2, Y_3, Y_4$

Now that we can see how to map a sequences of arbitrary length to more sequences of arbitrary length, what can we use it for?

1. Machine translation of languages

   train with parallel text

2. Speech recognition : sounds → words

   require lots of data to work well

3. Captions for images

   Convnets : map images to vector representations of the contents

   If we connect a convnet to an RNN we can map an image to a sequence of words - a caption

Final Project

$$r = r_0 \beta^{t/\tau}$$

$$= r_0 \left(\beta^{1/\tau}\right)^t$$

normally we'd use something like

$$r = r_0 e^{\alpha t}$$

So $\beta^{1/\tau}$ is like e^{α}