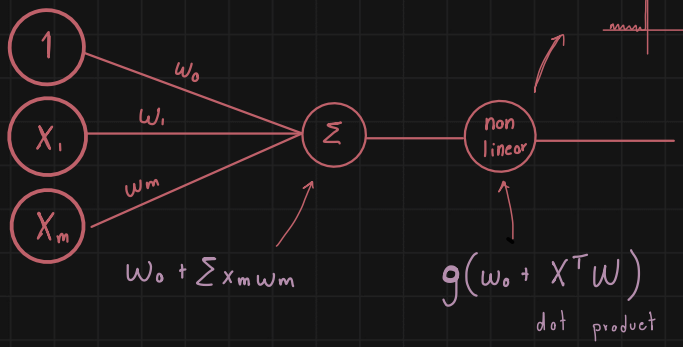
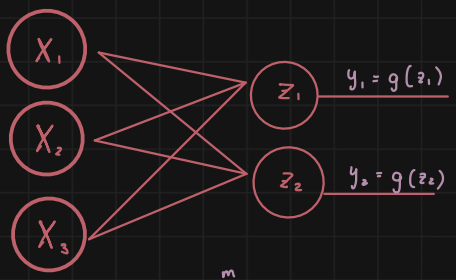
Lecture One

Perceptron

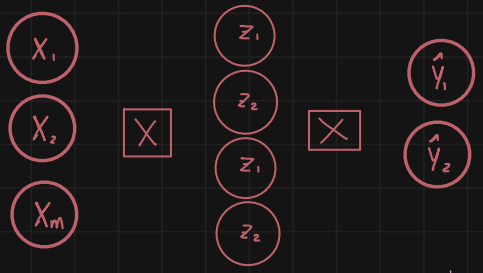


Dense Layer (multiple perceptrons)

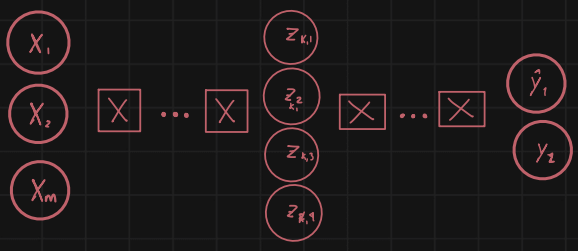


$$z_i = w_{0,i} + \sum_{j=1}^{m} x_j w_{j,i}$$

Single Layer



Deep Neural Network



$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k+1}} g(z_{k-1,j}) w_{j,i}^{(k)}$$

Loss:

Cost of incorrect predictions

$$\mathcal{L}(f(x^{(i)}; W), y^{(i)})$$

(predicted, real)

[if the difference between predicted and real is very high, the loss will be very high]

Empirical Loss

Total Loss of entire data set

$$X = \begin{bmatrix} 4,5 \\ 2,1 \\ \dots \end{bmatrix} \quad (x_1)(x_2) \boxtimes (z_1)(z_2)(z_3) \boxtimes (y_1) \quad f(x) \begin{bmatrix} 0.1 \\ 0.6 \end{bmatrix} \Big| \Big| \begin{bmatrix} 1 \\ 1 \end{bmatrix} y \quad \times \checkmark$$

$$J(W) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\text{predicted}, \text{actual})$$

(just the average)

Minimize loss

Mean Squared Error Loss

For continuous numbers as outputs

Binary Cross Entropy Loss

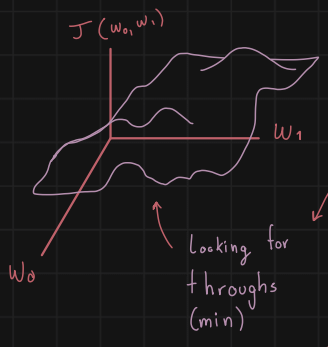
For binary (0,1) outputs

Loss Optimization

Find the weights for the lowest loss

$$W^* = \underset{W}{\operatorname{argmin}}\, J(W)$$

$$W = \{W_0, W_1, \dots\}$$

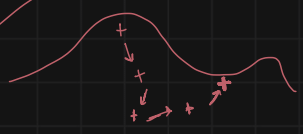


0. Take random point
1. and take gradient — AKA which way is up/down

$$\frac{\partial J(W)}{\partial W}$$

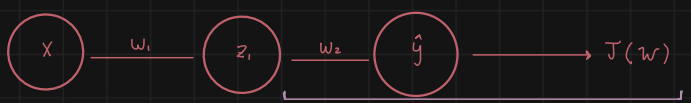
2. Take small step towards opposite of gradient — AKA to the min
3. Repeat until local (or global, we don't know) minimum — AKA update weights by small amount
4. Return weights

Gradient Descent



How to actually train NNs to get the precise weights (precise prediction)

How to compute gradient: Back propagation



Gradient of w_2

$$\frac{\partial J}{\partial w_2} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_2}$$

(small change in w_2 affects $J(w)$)

Gradient of w_1

$$\frac{\partial J}{\partial w_1} = \frac{\partial J}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1} \frac{\partial z_1}{\partial w_1}$$

(w_1 affects w_2 which in turn affects $\hat{y}$)

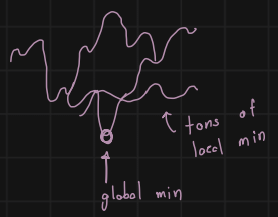
Back Propagation

(Repeat for every weight in the network)

[Apply chain rule over and over]

Training in Practice: Optimization

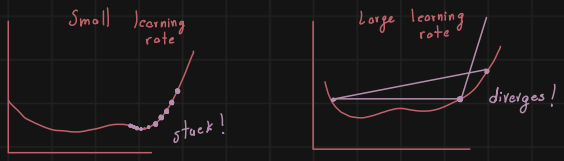
Training is extremely hard



Optimization of gradient descent

$$W \leftarrow W - \eta \frac{dJ}{dW}$$

Learning rate (How large every step should be)



We want stable learning rates
(converge smoothly but avoid local min)

We can
- Try a lot of learning rates and see which fits.
- Use adaptive learning rate
(Optimizers, for example: SGD, Adam, Adadelta, etc)

Training in Practice: Mini Batching

Gradient descent
A derivative is computationally intensive $\frac{dJ(W)}{dW}$ All weights (slow, precise)

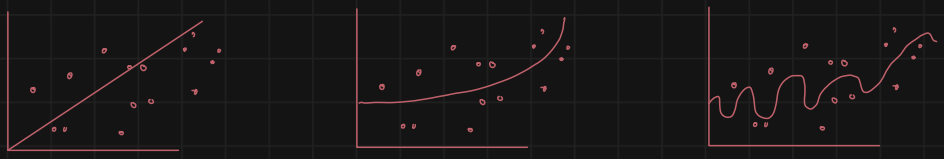
Stochastic Gradient descent
We can compute a single point, to be faster $\frac{dJ(w)}{w}$ Single point (fast, very noisy)

There is a middle ground

Batching
We can compute a number B of points $\frac{1}{B}\sum_{k=1}^{B}\frac{dJ_k(W)}{dW}$ Multi point (fast, kinda precise)

Mini-batches allow parallelization !!

Training in Practice: Overfitting

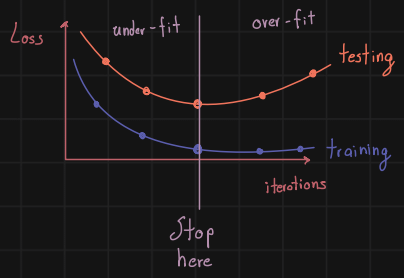


Regularization
Technique to discourage overfitting

- Dropout
Randomly set activations to 0 (lol)
- Forces to rely on 1 node

- Early Stopping

  stop before it can overfit

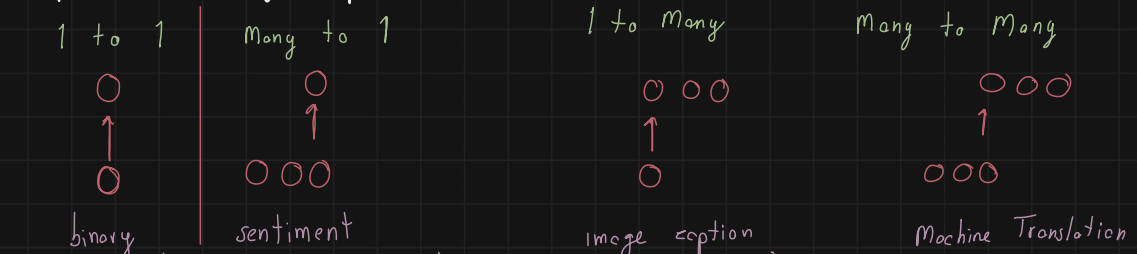


→ (will continue to go down because it is "memorizing" the data)

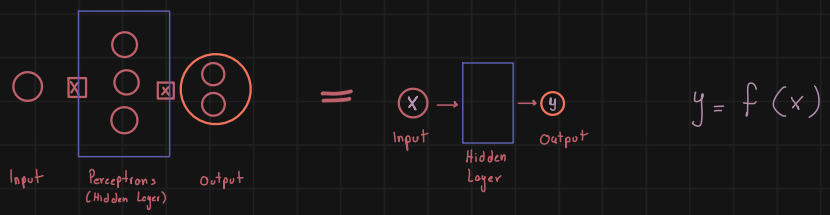
Lecture 2

Sequencial Modeling

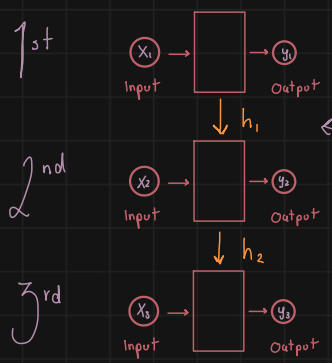
Sequence Modelling Applications



Neurons with Recurrence



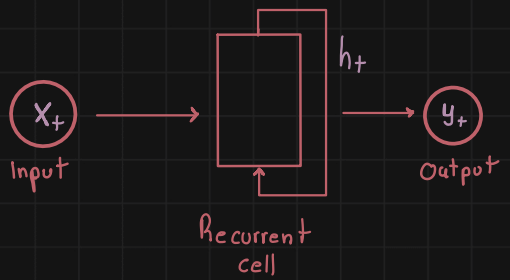
How to make this time dependent ?



So it goes from

$$y = f(x_t) \quad \text{to} \quad y = f(x_t, h_{t-1})$$

Now the output depends on the memory (previous calculation)



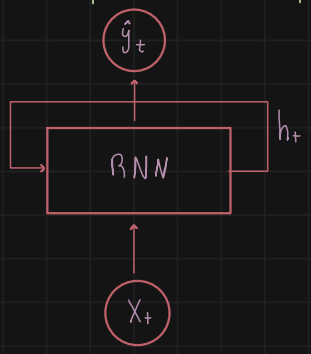
h_t : State

Is updated at each step

$$\underbrace{h_t}_{\text{cell state}} = \underbrace{f_W}_{\text{function weights}}(\underbrace{x_t}_{\text{input}}, \underbrace{h_{t-1}}_{\text{old state}})$$

Same function !

State update and Output

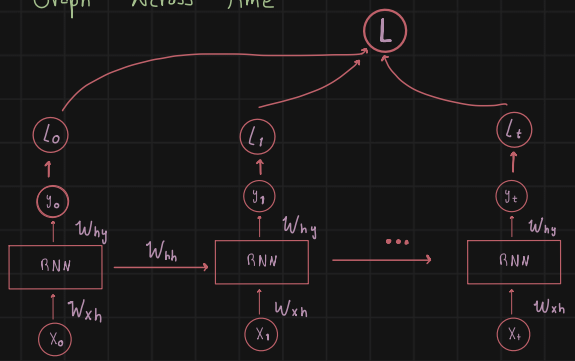


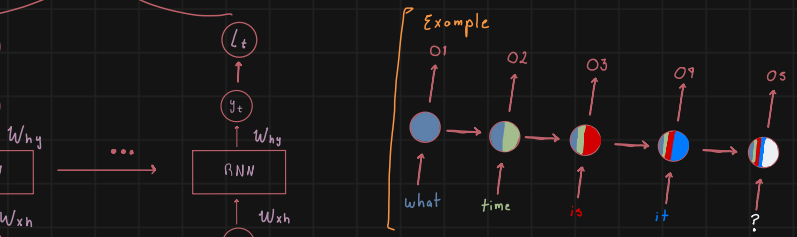
Output $\hat{y}_t = W_{hy}^T h_t$

Update hidden state $h_t = \tanh\left(W_{hh}^T h_{t-1} + W_{xh}^T X_t\right)$

Input vector X_t

RNN: Graph Across Time





3 different weights

Sequence Modelling: Design Criteria

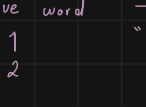
- Variable length
- Maintain order
- Share states (memory)
- Track long-term dependencies

Models work on numerical inputs, of course (math functions)

Embedding: Transform indexes into a vector

Example

1. Vocabulary
   "a" "dog" "took" "my" "I"
2. Indexing
   Value to every unique word
   a → 1
   dog → 2
   ...
3. Embedding
   One hot embedding
   "dog" = [0,1,0,...]
   index(2)
   (Binary index)
   Learn embedding

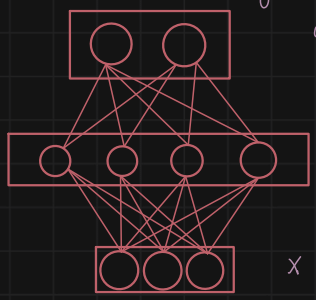


Feed index map to model.
Similar words will have similar embeddings

Back Propagation Through Time

How to train

Normal backpropagation:



First

Forward pass

makes a prediction

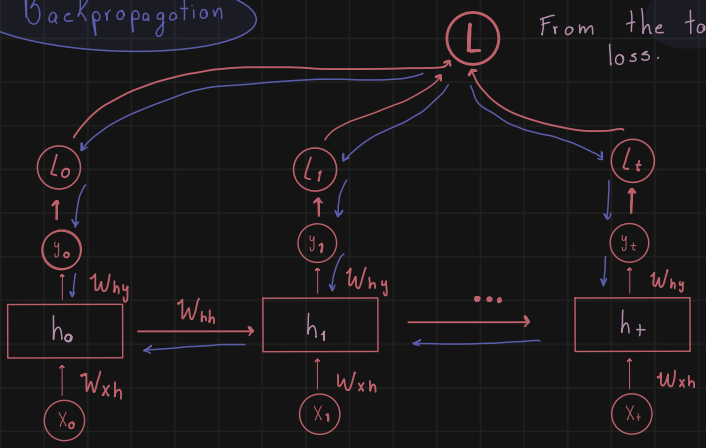
Then

Computes Loss (pred, truth)

To train, we back propogate:

1. Derivate loss with respect to each parameter
2. Shift parameters to minimize loss

Backpropagation



The gradient with respect to h_0 involves many factors of W_{hh} and repeated gradient calculation.

This is problematic

Problem 1: Exploding gradient

Many Values > 1

When we have a lot of values, gradients extremely large.

We use Gradient Clipping

Problem 2: Vanishing Gradients

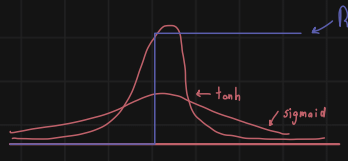
Many values < 1

When we have a gradient so small that we cannot train anymore.

Multiply many small numbers → When going back in time they get smaller and smaller → This adds a BIAS to the short-term dependencies

- Solution 1: Activation Function

  Use ReLU activation function



The $f' > 1$ when $x > 0$

- Solution 2: Weight Initialization

  Initolize weights to $\mathbb{I}$ matrix $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

  Initiolize biases to 0

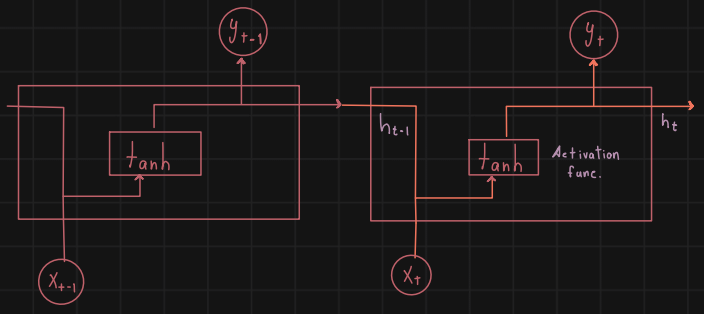
  Prevents weights from shrinking

- Solution 3: Network Architecture

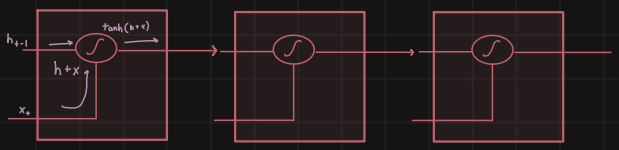
  Complex Recurrent Unit: Gated Cell

Gated Cell (LSTM, GRU, etc)

Controls which information goes through.



Vanilla Recurrent NN (Standard)



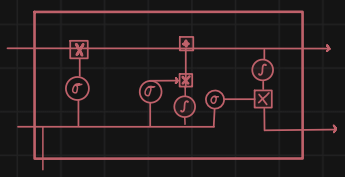
Vanishing gradient
(shrinks as backpropagates through time
Doesn't learn

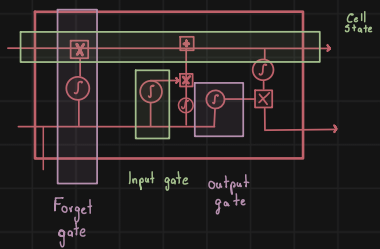
LSTM (Long Short Term Memory)

Mitigate short term memory (vanishing gradient)

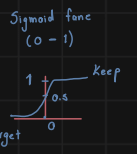
Rely on gated cell to track info throught steps

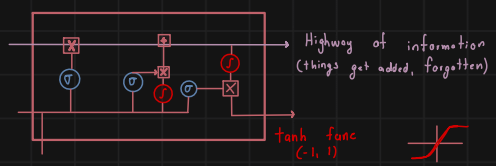
Gate: Regulate flow of information

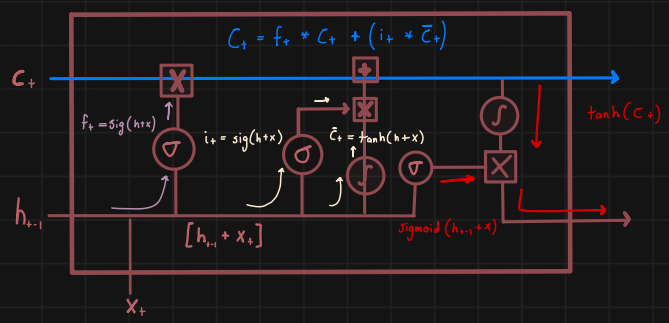




Description of Components







Forget gate

What to Keep (sigmoid (h+x) ≈ 1) or to forget (sigmoid (h+x) ≈ 0)

Input gate

Updates the state. The sigmoid determines which value is importon (1: important, 0: Not important) and then gets multiplied by tanh to regulate. Sig decides which info to keep from tanh.

New Cell State

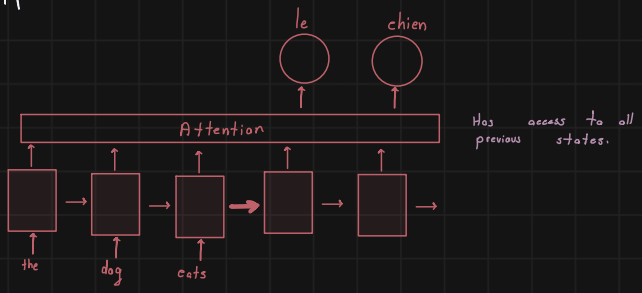
Previous state multiplied by forget vector, added to the input gate selection (update values)

Output gate

Previous hidden state / input into sigmoid and then multiply with tanh of new cell state to produce a new hidden state.

RNN Applications

Google translate
(That's why you get better contextual translation with more words)



Lecture 3

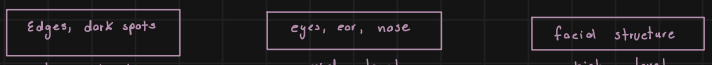
Convolutional Neural Networks

Computer vision (context)

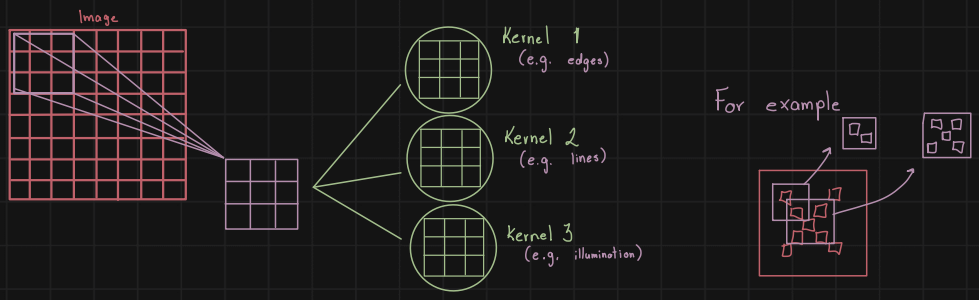
Learning Feature Representations

Lear from a hierarchy of features directly from data.

for example



Learning Visual Features



Convolution Operation

ex. 1

Input		⊗	Filter		=			sum →	
3	2		1	-1		3	-2	6 - 9 =	2
2	3		-1	1		-2	3		

ex 2.

1	2	3	⊗	2	2	=	1	9	9	6	sum →
1	2	3		1	1		1	2	2	3	
1	2	3					1	9	9	6	
							1	2	2	3	

8	15
8	15

So, as coded on github, the ove several filters

like

0	-1	0
-1	5	-1
0	-1	0

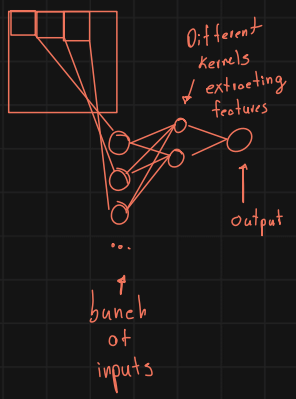
sharpen

0	1	0
1	-4	1
0	1	0

edge detect

but CNN get these values themselves !

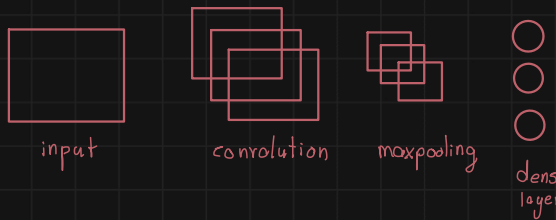
simply put



So :

- Apply weights (filter / kernel) to extract local features
- Use multiple filters to get multiple features
- Spatially share parameters of each filter

CNN for classification



1. Convolution
2. Non linearity
3. Pooling (down-sampling)

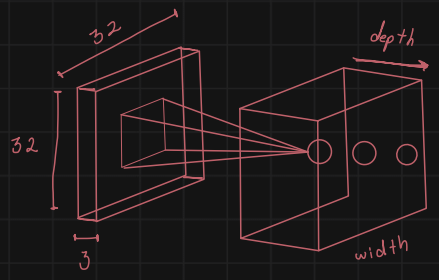
Each neuron sees a patch of its inputs

Apply Kernel

Compute linear combination

$$\text{non linear}\left(\sum_{i=1}^{4}\sum_{j=1}^{4} W_{ij}\, X_{i+p,\, j+q} + b\right)$$

Finally, activate with non-linear function



depth is the number of filters

Usually non-linear operation is Relu

negative returns 0
positive stays the same

Max pooling reduces dimensionality

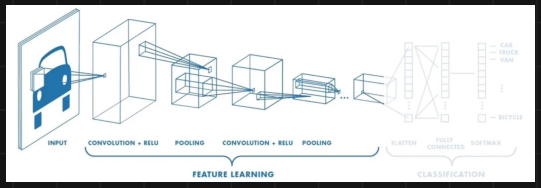
Divide into 2x2 patches

1	1	3	4
2	8	3	10
5	5	20	1
1	19	1	1

take the max value →

8	10
19	20

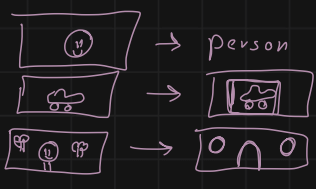
- Mean pooling takes the average
- Min the minimum ...




```python
import tensorflow as tf

def generate_model():
    model = tf.keras.Sequential([
        # first convolutional layer
        tf.keras.layers.Conv2D(32, filter_size=3, activation='relu'),
        tf.keras.layers.MaxPool2D(pool_size=2, strides=2),

        # second convolutional layer
        tf.keras.layers.Conv2D(64, filter_size=3, activation='relu'),
        tf.keras.layers.MaxPool2D(pool_size=2, strides=2),

        # fully connected classifier
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(1024, activation='relu'),
        tf.keras.layers.Dense(10, activation='softmax')  # 10 outputs
    ])
    return model
```

Cool Applications

Classification	What is this image?	→ person
Object detection	Where are the objects?	→
Segmatic segmentation	What class this belongs?	→



Lecture 9

Deep Generative Models

Supervised Learning — Data (x, y) — y: Label

Function to map $x \rightarrow y$

vs

Unsupervised Learning — Data (x) — No labels!

Learn the structure of the data

Generative Modelling

Input training for some distribuiton and learn a model to represent that distribuiton.

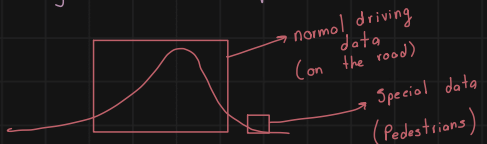
Use cases:

Debiasing

Uncovering underlying features to make the data more distributed

Outlier Detection

Detect very rare and unique cases in training data

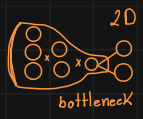


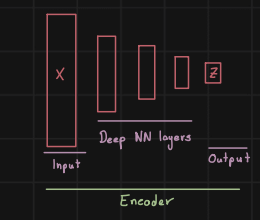
Latent Variable Models

true explanatory factors

Autoencoders

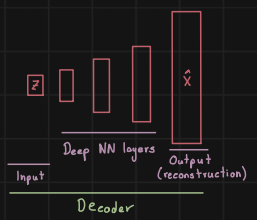
Learn a lower dimensional feature of raw data





(maps x to a lower dimensional space z)

$x \rightarrow z$



(maps back from z to a constructed observation $\hat{x}$)

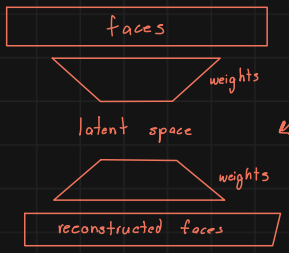
$z \rightarrow x$

To train, we compare x with $\hat{x}$

e.g. $L(x, \hat{x}) = |x - \hat{x}|^2$ (mean squared error)

Loss doesn't use any labels!?

You could train a model

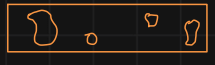


When trained just input latent vector and get a entirely new face

But what does each dimension on the vector mean ???

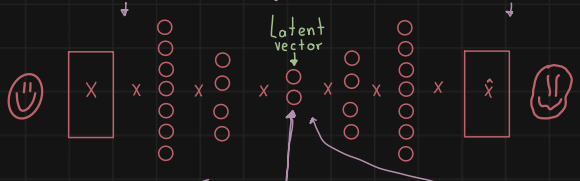
$\begin{bmatrix} v \\ y \\ s \end{bmatrix}$ → inclination ? → skin tone → eye color

We don't Know!



Pool too wide

What comes through input is reconstructed in output



Forces the network to learn a representation of data very small

i.e. all info to reconstruct is in 2 neurons

Like really, really good compression

Variational Autoencoders (VAEs)

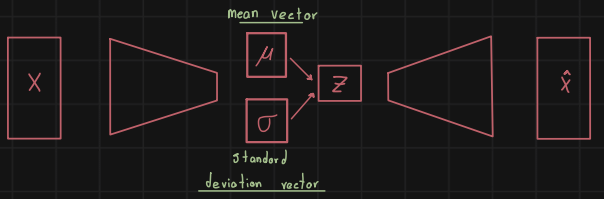
Now we CAN create new things because we know the distribution

our imposed distribution

Traditional autoencoders are deterministic

input → weights → reconstructed output

But VAEs



Replace deterministic nature

input → mean, variance → reconstructed output

Encoder (computes $q_\phi(z|x)$) | Decoder (computes $p_\theta(x|z)$)

separate weights

We train by

$$\mathcal{L}(\phi, \theta, x) = (\text{reconstruction loss}) + (\text{regularization term})$$

reconstruction loss: input vs reconstructed output

eg. $|x - \hat{x}|^2$

regularization term (minimize encoder and prior): $D(q_\phi(z|x) \| p(z))$

$q_\phi(z|x)$: prob dist of z given x (encoder); $p(z)$: prior (hypothesis)

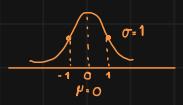
$$D(q_\phi(z|x) \| p(z))$$

$p(z)$: Prior

Normally Normal - Gaussian Distributions

$$p(z) = \mathcal{N}(\mu = 0, \sigma^2 = 1)$$

↑ sort of force the distribution to have



sample from $\mu + \sigma \odot \varepsilon$