

MIT Introduction To Deep Learning

6S.191

What is Deep Learning

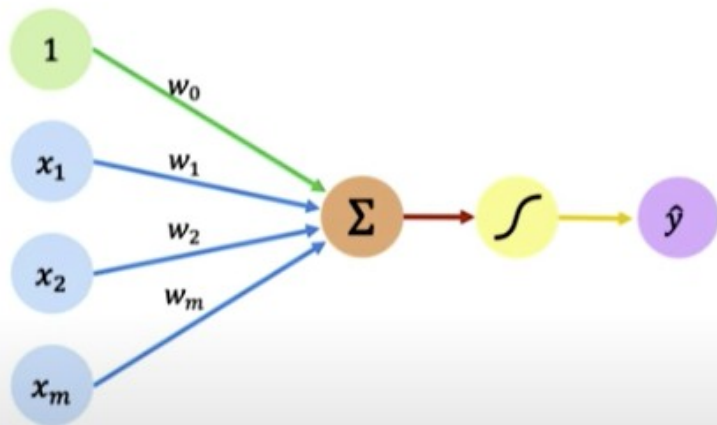
- Intelligence
 - Ability to process information and make informed future predictions
 - Artificial Intelligence
 - Any technique that enables computer to mimic human behavior
 - Machine Learning
 - Ability to learn without explicitly being programmed
 - Deep Learning
 - Use of Neural Networks
 - Extract Patterns or Features from data using neural networks to inform those decisions
 - Given a bunch of examples (dataset) how can we teach the computer to complete the task
-

Why Deep Learning and Why Now

- Traditional Machine Learning
 - define a set of rules or features in the environment of the data
 - Hand Engineered features are time consuming and not scalable in practice
- In Deep Learning
 - The features are going to be learned from the data itself in a hierarchical manner
 - Given a data set (detect faces) can we train a deep learning model
 - input model is a face
 - Can we detect low level features such as edges building up those edges to create (eyes, noses, mouth) mid level features and then build those features into larger features such as the face (High Level Features)
 - As you go deep into a neural work you see it will capture those type of features
 - Can we learn the underlying features directly from data
 - As you research deeper into a neural network you will see its ability to capture that hierarchical data
 - The ability to extract the features and perform machine learning on them. – Goal of Deep learning
- Why Now
 - Data has become prevalent
 - Hardware are massively parallelizable
 - Open Source of Software

The Perceptron : Forward

The Perceptron: Forward Propagation



$$\hat{y} = g \left(w_0 + \sum_{i=1}^m x_i w_i \right)$$

$$\hat{y} = g (w_0 + \mathbf{X}^T \mathbf{W})$$

$$\text{where: } \mathbf{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \text{ and } \mathbf{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$

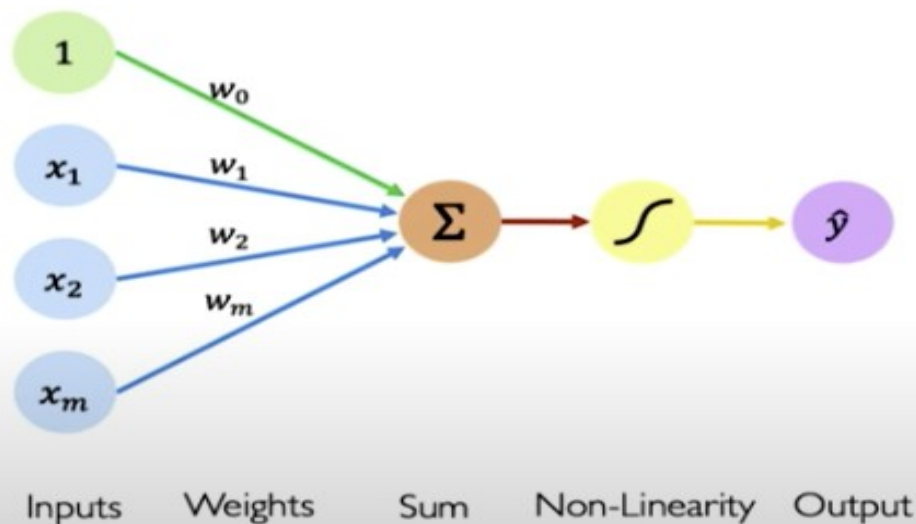
Inputs Weights Sum Non-Linearity Output

The Perceptron : Forward Propagation

- Input on the Left and outputs on the right
- The Linear Combination of Inputs is passed through a Non Linear Function called an Activation Function
- The output will be our prediction
- We have a bias term which is called w_0 which shifts the input of the activation to the left or right
- The equation can be rewritten using linear algebra in terms of vectors and dot products

The Perceptron : Forward Propagation

The Perceptron: Forward Propagation



Activation Functions

$$\hat{y} = g(w_0 + \mathbf{X}^T \mathbf{W})$$

- Example: sigmoid function

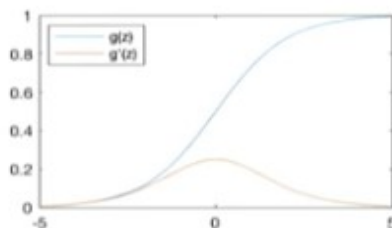
$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



The Perceptron : Forward Propagation


Common Activation Functions

Sigmoid Function

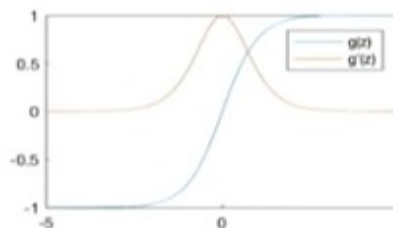


$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$


 `tf.math.sigmoid(z)`

Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

 `tf.math.tanh(z)`

Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

 `tf.nn.relu(z)`

 TensorFlow code blocks

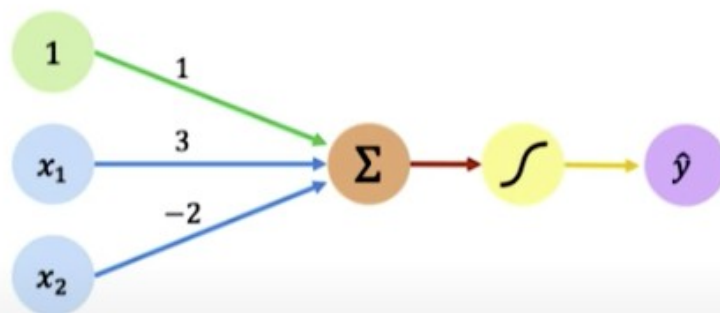
NOTE: All activation functions are non-linear

The Perceptron : Forward Propagation

- Sigmoid Function
 - Popular since it outputs values between 0 and 1
 - model probabilities
- Relu Function
 - single non linearity with $x = 0$
 - Rectified Linear Unit
- Activations Function
 - Introduce Non Linearity into the network
 - Example – Separate the colors by a line without curving it, – However, only a curve will separate the colors
 - If you can only use a linear activation function the problem is impossible (add a line to a line the result is a line)
 - Non Linearities allow use to approximate arbitrary complex functions and that make it powerful
 - The Data usually will be non linear and take the form of a curve
 -

The Perceptron Example

The Perceptron: Example



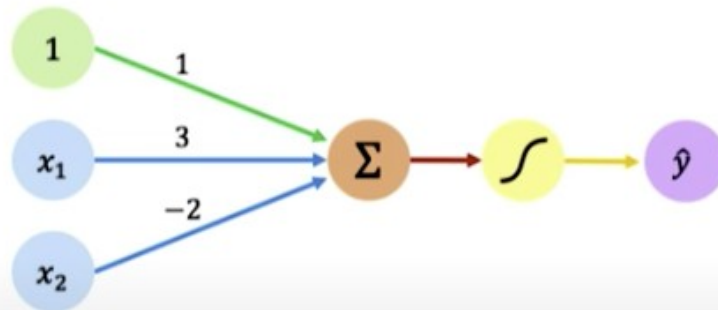
We have: $w_0 = 1$ and $\mathbf{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\begin{aligned}\hat{y} &= g(w_0 + \mathbf{X}^T \mathbf{W}) \\ &= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\ \hat{y} &= g(1 + 3x_1 - 2x_2)\end{aligned}$$

This is just a line in 2D!

The Perceptron Example

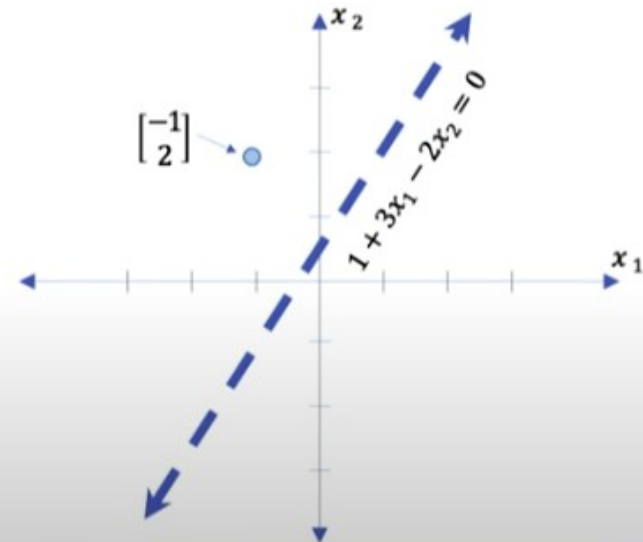
The Perceptron: Example



Assume we have input: $\mathbf{x} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

$$\begin{aligned}\hat{y} &= g(1 + (3 * -1) - (2 * 2)) \\ &= g(-6) \approx 0.002\end{aligned}$$

$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



The Perceptron Example

- Notes from preceding slide
 - Can plot this line in feature space
 - The line is the decision boundary
 - defines the perceptron neuron
 - If we are give a point we can see where it is in 2D space in regards to the line
- For $x = 0$ then the value of y will be less then .5 and for $x > 0$ the value of $y > .5$
- Cannot draw these graphs for neural networks since there are many inputs and not just 2 values and a weight.
-

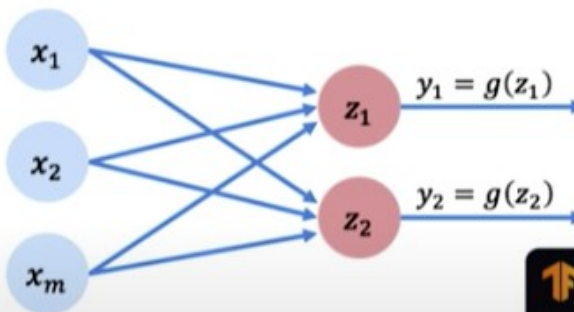
From perceptrons to neural networks

- Each perceptron is made up of a bias, dot product (input and weights) and provide a bias
- All inputs connected to all outputs these layers are called dense layers

Multi Output Perceptron

Multi Output Perceptron

Because all inputs are densely connected to all outputs, these layers are called **Dense** layers



```
import tensorflow as tf  
layer = tf.keras.layers.Dense(  
    units=2)
```

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

Dense Layer from Stratch

Dense layer from scratch



```
class MyDenseLayer(tf.keras.layers.Layer):
    def __init__(self, input_dim, output_dim):
        super(MyDenseLayer, self).__init__()

        # Initialize weights and bias
        self.W = self.add_weight([input_dim, output_dim])
        self.b = self.add_weight([1, output_dim])

    def call(self, inputs):
        # Forward propagate the inputs
        z = tf.matmul(inputs, self.W) + self.b

        # Feed through a non-linear activation
        output = tf.math.sigmoid(z)

        return output
```

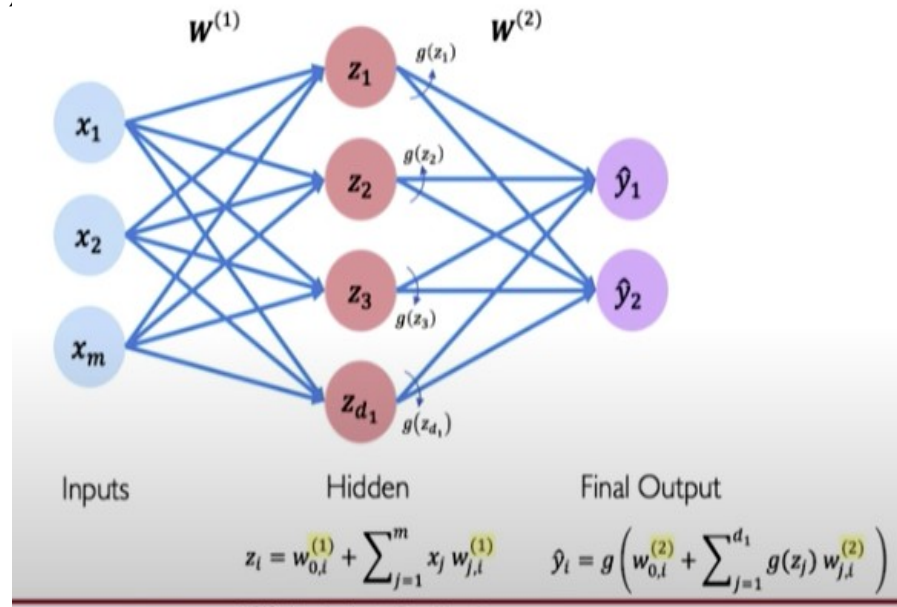
Single Hidden Layer Neural Network

- The hidden layers are not directly observable or enforceable such as the input and the output we now we want to predict
- Need two sets of matrices (Input Layer to hidden layer, hidden layer to Output Layer)



Symbol for a dense Layer

- Fully connected Layers – Another name for dense layer
- Sequential Model feed inputs sequentially from inputs to inputs
- Dense Network
 - Stack Dense layers into a sequential model –
 - Where the inputs are fed sequentially to the input to outputs
 - Example
 - `model = tf.keras.Sequential (`
 - `[tf.keras.layers.Dense(n), tf.keras.layers.Dense(2)]`
- Deep Learning Networks
 - Keep stacking hidden layers to create hierarchical neural network
 - The output is going deep and deeper into the neural network.



Applying Neural Networks

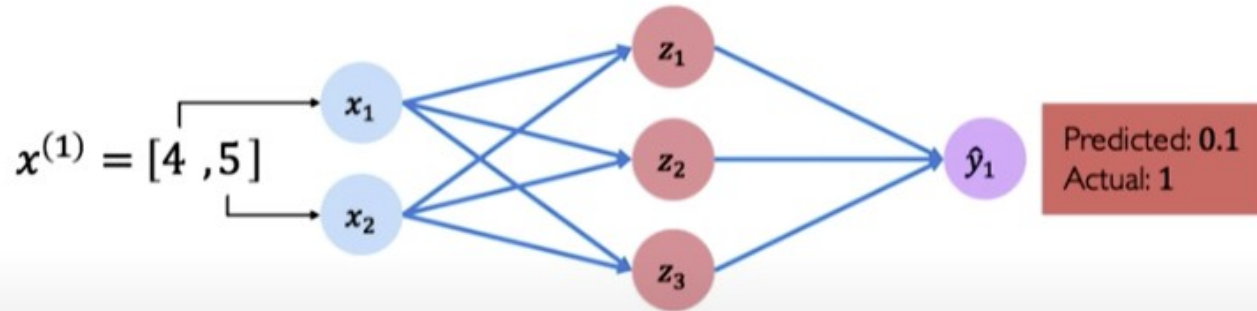
Loss Functions

- Will I pass the class
- Lets start with simple two feature model
 - x_1 – Number of lectures you attend
 - x_2 – Hours spent on the final project
- Data – Can collect data from how each student performed here
- Build a neural network to determine will I pass or fail that class
- Why was the predicted .1 for values that should have passed the class
 - Problem the network was never trained
 - Has no idea about the data
 - Define Loss
 - Empirical want to minimize the loss over the whole dataset
 - uses the mean
 - Using : `softmax_cross_entropy_loss` – for classification
 - If we changed the output to the numeric grade then we might use the MSE Loss function instead

Applying Neural Networks

Quantifying Loss

The **loss** of our network measures the cost incurred from incorrect predictions



$$\mathcal{L}(\underbrace{f(x^{(i)}; \mathbf{W})}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}})$$

The letter that looks like an L in script is the loss function

Loss Functions

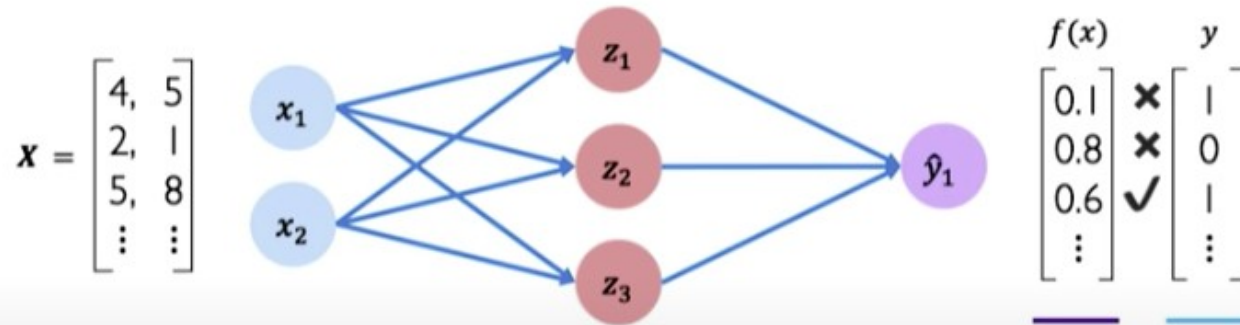
The closer the prediction is to the actual value the less error that will happen

Mean of the individual loss functions of the dataset

Loss Function

Binary Cross Entropy Loss

Cross entropy loss can be used with models that output a probability between 0 and 1



$$J(W) = -\frac{1}{n} \sum_{i=1}^n \underbrace{y^{(i)}}_{\text{Actual}} \log \left(\underbrace{f(x^{(i)}; W)}_{\text{Predicted}} \right) + (1 - \underbrace{y^{(i)}}_{\text{Actual}}) \log \left(1 - \underbrace{f(x^{(i)}; W)}_{\text{Predicted}} \right)$$



```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(y, predicted) )
```

Training Neural Networks

Loss Optimization

We want to find the network weights that *achieve the lowest loss*

$$W^* = \operatorname{argmin}_W \frac{1}{n} \sum_{i=1}^n \mathcal{L}(f(x^{(i)}; W), y^{(i)})$$

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

W is the collection of weight in the neural network.

W^* - A set pf weights that will provide the minimum loss average on a data set

Training Neural Networks

- Want to find a set of weights W^*
 - A set of weights that give use the minimum loss function on the dataset
 - This is one of the goals of creating a neural net.
- W is group of all the weights from every layer in the model
- The output of the loss function is the weights

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(W)}{\partial W}$
4. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights

Once we do step 4 , we will take a small step in the opposite direction

- The amount we step here determines the magnitude of the step
- convergence – Where the convergence of neural networks – Learning rate becomes lower and the errors produced by the model in training comes to a minimum

Training Neural Networks

```
import tensorflow as tf

weights = tf.Variable([tf.random.normal()] )

while True:    # loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)

    weights = weights - lr * gradient
```

Back Propagation



How does a small change in one weight (ex. w_2) affect the final loss $J(W)$?

We decompose the left side using the chain rule and decompose into two terms. We have access to the two gradients on the right hand side. \hat{y} is only dependent on the previous layer. We want to compute the gradients on w_1 then we do it recursively. We repeat this process for every weight so we can determine how the weight need to change on decrease the loss on the next iteration

Back Propagation

$$\frac{\partial J(W)}{\partial w_2} = \underbrace{\frac{\partial J(W)}{\partial \hat{y}}}_{\text{purple}} * \underbrace{\frac{\partial \hat{y}}{\partial w_2}}_{\text{red}}$$

Repeat this process for every weight in the network using gradients from later layers. Then we can determine how every single weight needs to change and how they need to change to decrease our loss on the next iteration

We can apply the changes so our loss is a little bit better on the next trial

Compute the gradient and step in the opposite direction

Apply the Chain Rule Recursively

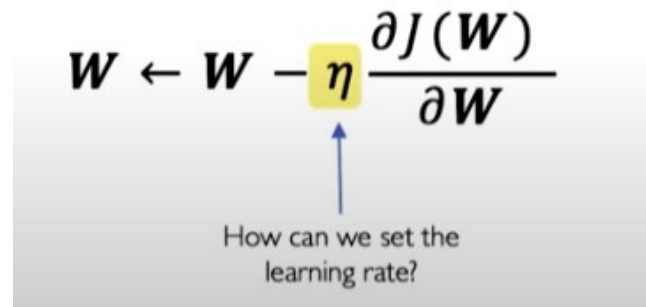
$$\frac{\partial J(W)}{\partial w_1} = \underbrace{\frac{\partial J(W)}{\partial \hat{y}}}_{\text{purple}} * \underbrace{\frac{\partial \hat{y}}{\partial z_1}}_{\text{red}} * \underbrace{\frac{\partial z_1}{\partial w_1}}_{\text{blue}}$$

Apply chain rule! Apply chain rule!

$$\frac{\partial J(W)}{\partial w_1} = \underbrace{\frac{\partial J(W)}{\partial \hat{y}}}_{\text{purple}} * \underbrace{\frac{\partial \hat{y}}{\partial z_1}}_{\text{red}} * \underbrace{\frac{\partial z_1}{\partial w_1}}_{\text{blue}}$$

Loss Functions can be difficult to Optimize

- η is the learning rate
- Optimization through
 - gradient descent
- How much step and trust we take
 - If we set the learning to be very slow then it may get stuck in a local minimum



The diagram shows the gradient descent update rule:
$$W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$$
 The learning rate η is highlighted in a yellow box. A blue arrow points from the text "How can we set the learning rate?" below to the yellow box.

Neural Networks in Practice: Optimization

- Training Neural Networks is difficult
- Gradient Descent
 - Lets look at the learning rate
 - Learning Rate – Determine the step or the trust we take in the gradients
 - The direction we need to go to minimize the loss – so go in the opposite direction
- Small learning rate converges slowly and get stuck in false local minima
- Large learning rates overshoot become unstable and diverge
- Stable learning rates converge smoothly and avoid local minima
- How do we find our learning rate
 - Try lots of different learning rates and see what works “just right”
 - Common technique
 - Design an adaptive learning rate that adapts to the landscape.
 - Smarter than the first idea
 - Learning rates are no longer fixed
 - Can be made larger or smaller depending on
 - how large the gradient is
 - how fast learning is happening
 - size of particular weight etc...

$$W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$$

How can we set the learning rate?

Compute gradient, $\frac{\partial J(W)}{\partial W}$

The Gradient is the slope

Optimization

Gradient Descent Algorithms

Algorithm	TF Implementation	Reference
• SGD	 <code>tf.keras.optimizers.SGD</code>	Kiefer & Wolfowitz. "Stochastic Estimation of the Maximum of a Regression Function." 1952.
• Adam	 <code>tf.keras.optimizers.Adam</code>	Kingma et al. "Adam: A Method for Stochastic Optimization." 2014.
• Adadelata	 <code>tf.keras.optimizers.Adadelata</code>	Zeiler et al. "ADADELTA: An Adaptive Learning Rate Method." 2012.
• Adagrad	 <code>tf.keras.optimizers.Adagrad</code>	Duchi et al. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." 2011.
• RMSProp	 <code>tf.keras.optimizers.RMSProp</code>	

Additional details: <http://runder.io/optimizing-gradient-descent/>

Optimization

Putting it all together

```
import tensorflow as tf

model = tf.keras.Sequential([...])

# pick your favorite optimizer
optimizer = tf.keras.optimizer.SGD()

while True: # loop forever

    # forward pass through the network
    prediction = model(x)

    with tf.GradientTape() as tape:
        # compute the loss
        loss = compute_loss(y, prediction)

    # update the weights using the gradient
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables)))
```

Neural Networks in Practice : Mini-Batches

- The Gradient Descent is very computationally expensive because it computes the weights over the entire dataset
 - Computed as a summation of your dataset (The dataset could have billion/trillions elements)
 - May not be able to compute on every training iteration or epoch

- Solution

- Define a new gradient it computes it on a single random sample
 - very noisy estimate (stochastic)
- Pick a random small subset of the collection of training data
- Provides an estimate not an exact answer
 - Default batch size is 32 to 100

- More accurate estimation of gradient

- Smoother convergence allows for larger learning rates
- converge much quicker in practice

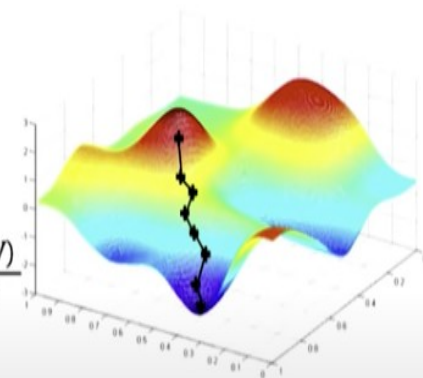
- Mini Batches lead to fast training

- Can parallelize computation
- achieve significant speed increases on GPUS
- Over each of the batch we can parallelize and get the avg gradient

Stochastic Gradient Descent

Algorithm

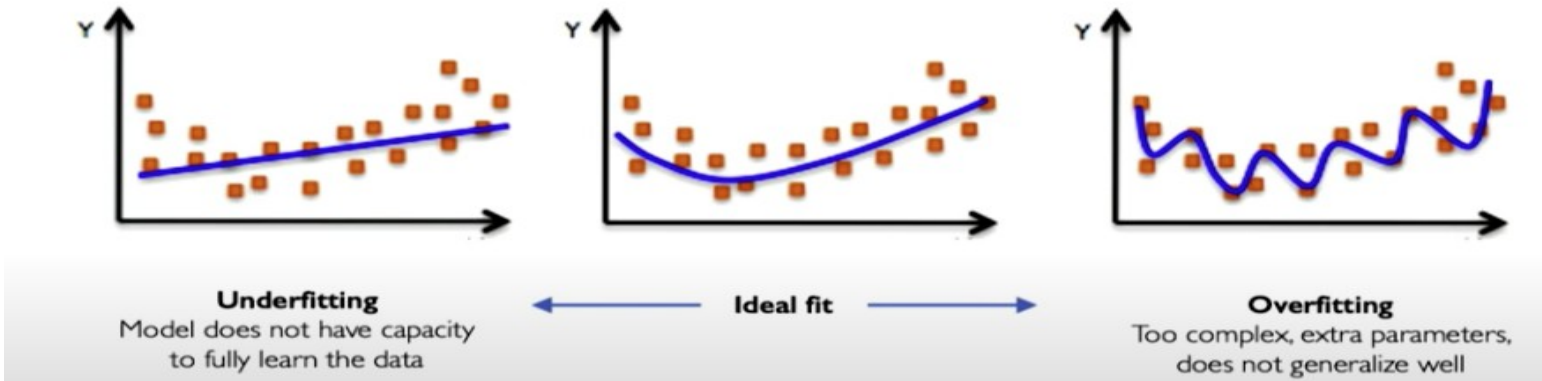
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(W)}{\partial W} = \frac{1}{B} \sum_{k=1}^B \frac{\partial J_k(W)}{\partial W}$
5. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
6. Return weights



The problem of overfitting

- Give a data set we want to learn the model that learns representations of our data that can generalize to new data
- We want a curve that is small enough to maintain generalize and large enough to capture the overall trends
- Regularization
 - Techniques that constrains our optimization problem to discourage complex models
 - Why do we need it – improve generalization of our model on unseen data

The Problem of Overfitting



The problem of overfitting

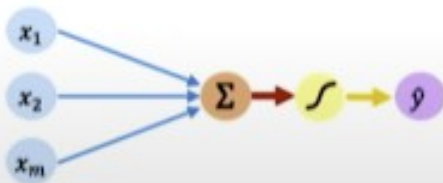
- Regularization Technique 1 (Drop Out) – During training randomly set some activations to 0
 - Typically drop 50% of activations in layer
 - Forces network to not rely on any one node
 - Identify different types of pathways through the network
 - Identify different forms of processing its information
 - In Tensorflow – `tf.keras.layers.Dropout(p=.5)`
- Regularization Technique 2 (Early Stopping) – Stop training before we have a chance to overfit
 - The training can be used to see when we start to overfit. (split it into two training sets)
 - The training and test loss keep going down until they diverge
 - Identify where the testing loss is minimized
 - capture where the testing loss is minimized for both the testing and training
 - Identify the place where the testing loss is minimized and that is going to be the model. that we are going to use



Summary

The Perceptron

- Structural building blocks
- Nonlinear activation functions



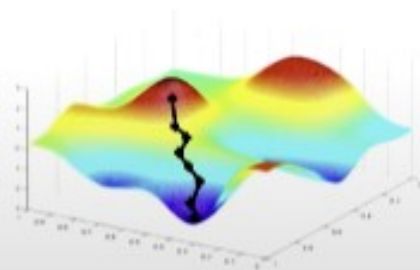
Neural Networks

- Stacking Perceptrons to form neural networks
- Optimization through backpropagation



Training in Practice

- Adaptive learning
- Batching
- Regularization



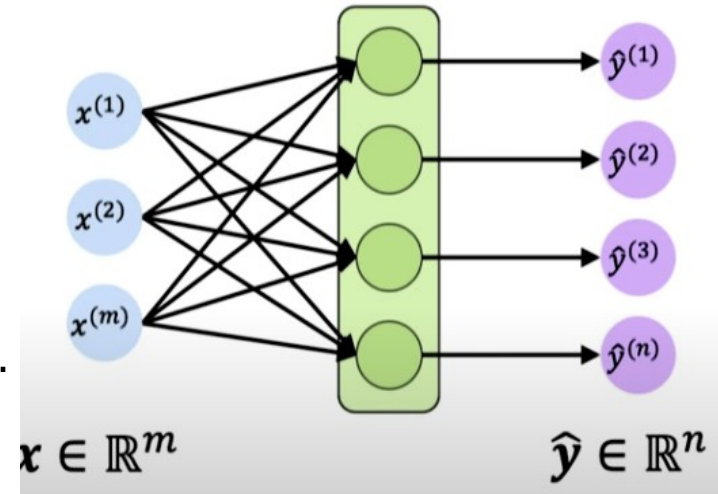
Deep Sequence Modeling

- Example
 - Given an image of a ball can you predict where it will go next.
 - need previous information to know where the ball will go next
- Example
 - Audio – Waveform of the voice can be split up into a sequence of sound waves
 - Text – Sequence of character or words
 - Medical – EKG Learning and Stock Prices / DNA Sequence
- Adds a component of time, we can handle different problems
- Example – Sentiment Classification – Many to One
 - Sequence of temporal inputs and get an sequential output
 - ex classify the emotion or sentiment of the tweet – map the sequence of words to a positive or negative label
- Example Image Captioning or text Generation – One to many – Image Captioning
 - The input may not have a time dimension The output has a temporal or sequential component (want to add some associate text)
- Example Machine Translation/Forecasting/Music Generation – Many to Many
 - Sequential Input mapped to a sequential output

Neurons with Recurrence

- Start with the perceptron from last layer
- In this example 3 inputs predict 4 outputs with a single layer of perceptrons. No idea of time or sequence yet.

One solution is could feed in a sequence to a model. For each time step fee it to the model. We are treating each time step as an isolated time step.



Issue: Don't have any connection between the the time steps

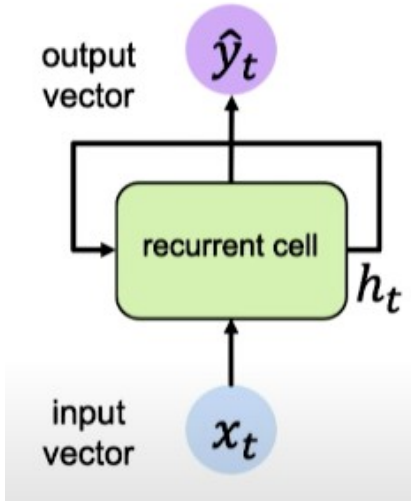
$$\hat{\underline{y}}_t = f(\underline{x}_t, \underline{h}_{t-1})$$

output input past memory

What we need is to relate the network computations at a particular time step to its prior computations. Link the computation of the network at different time steps to each other via a recurrence relation (A sequence is an equation based on some rule).

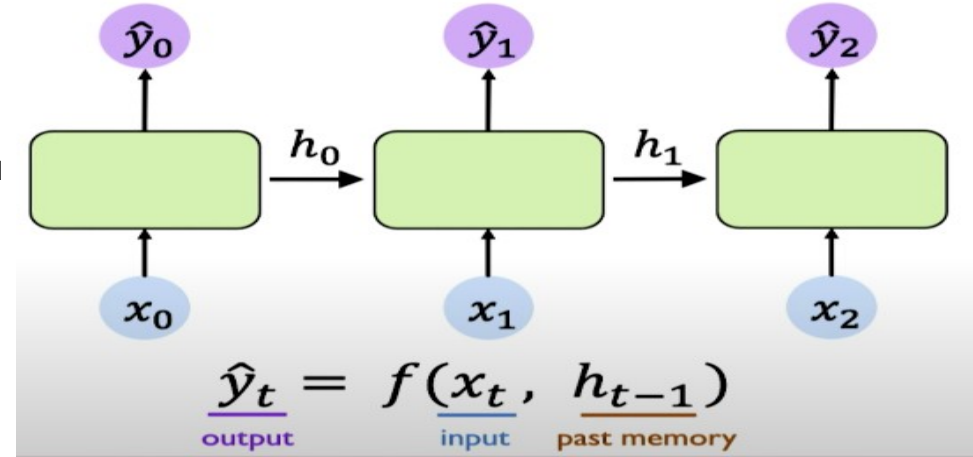
Neurons with Recurrence

- have an internal memory or state which is denoted as h_0, h_1
 - can be passed forward across time
- Not the function is dependent on the input and the previous state that is past forward
 - The individual time steps



Shows the relation as a cycle.

The RNN (recurrent cell) is represented as a computational graph unrolled across time.



Recurrent Neural Networks (RNN)

- RNNs have a state h_t that is updated at each time steps as a sequence is processed
- Apply a recurrence relation at every time step to process a sequence
- The set of weight is what the model is going to be learning through the network by
 - training
 - The weights are the same across all timestamps
 - The recurrence function is also the same
- Pseudo Code

```
my_rnn = RNN()
hidden_state = [0, 0, 0, 0]

sentence = ["I", "love", "recurrent", "neural"]

for word in sentence:
    prediction, hidden_state = my_rnn(word, hidden_state)

next_word_prediction = prediction
# >>> "networks!"
```

$$\boxed{h_t} = \boxed{f_W}(\boxed{x_t}, \boxed{h_{t-1}})$$

cell state function with weights W input old state

Task : Try to predict the next word that will come at the end of the sentence.

Recurrence relation is the for word in sentence. The prediction is the next word

hidden_state is set to distinct values

Recurrent Neural Networks (RNN)

- How do we derive the output vector
 - The function is tanh is the applied linearity

Output Vector

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

Update Hidden State

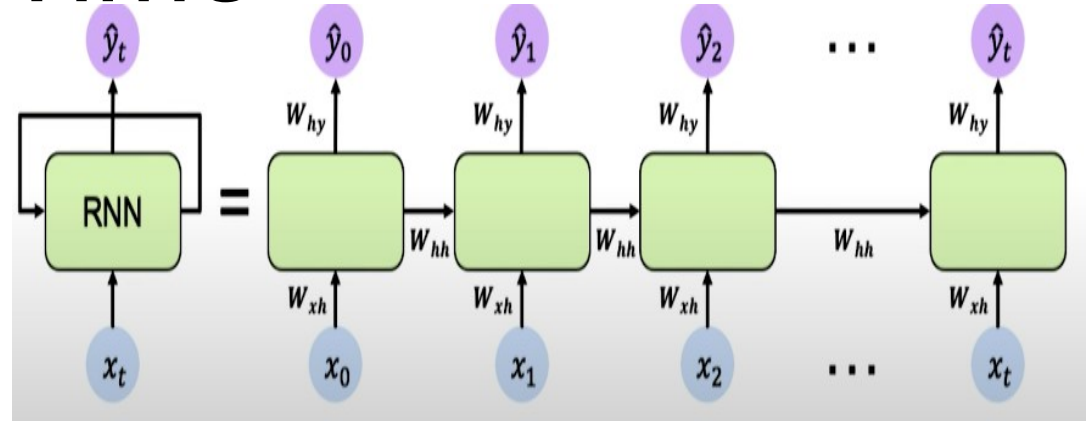
$$h_t = \tanh(W_{hh}^T h_{t-1} + W_{xh}^T x_t)$$

Input Vector

$$x_t$$

RNNs: Computational Graph Across Time

- We can have predictions at each individual time step
- The weight matrices are reused across all the time steps
- How do we train the neural network
 - The loss function will be a loss generated at every time step which is summed together to generate a total sum loss by summing them all together.
 - Do the loss function when we are making a forward pass through the network.



RNNs: Computational Graph Across Time

```
class MyRNNCell(tf.keras.layers.Layer):
    def __init__(self, rnn_units, input_dim, output_dim):
        super(MyRNNCell, self).__init__()

        # Initialize weight matrices
        self.W_xh = self.add_weight([rnn_units, input_dim])
        self.W_hh = self.add_weight([rnn_units, rnn_units])
        self.W_hy = self.add_weight([output_dim, rnn_units])

        # Initialize hidden state to zeros
        self.h = tf.zeros([rnn_units, 1])

    def call(self, x):
        # Update the hidden state
        self.h = tf.math.tanh( self.W_hh * self.h + self.W_xh * x )

        # Compute the output
        output = self.W_hy * self.h

        # Return the current output and hidden state
        return output, self.h
```

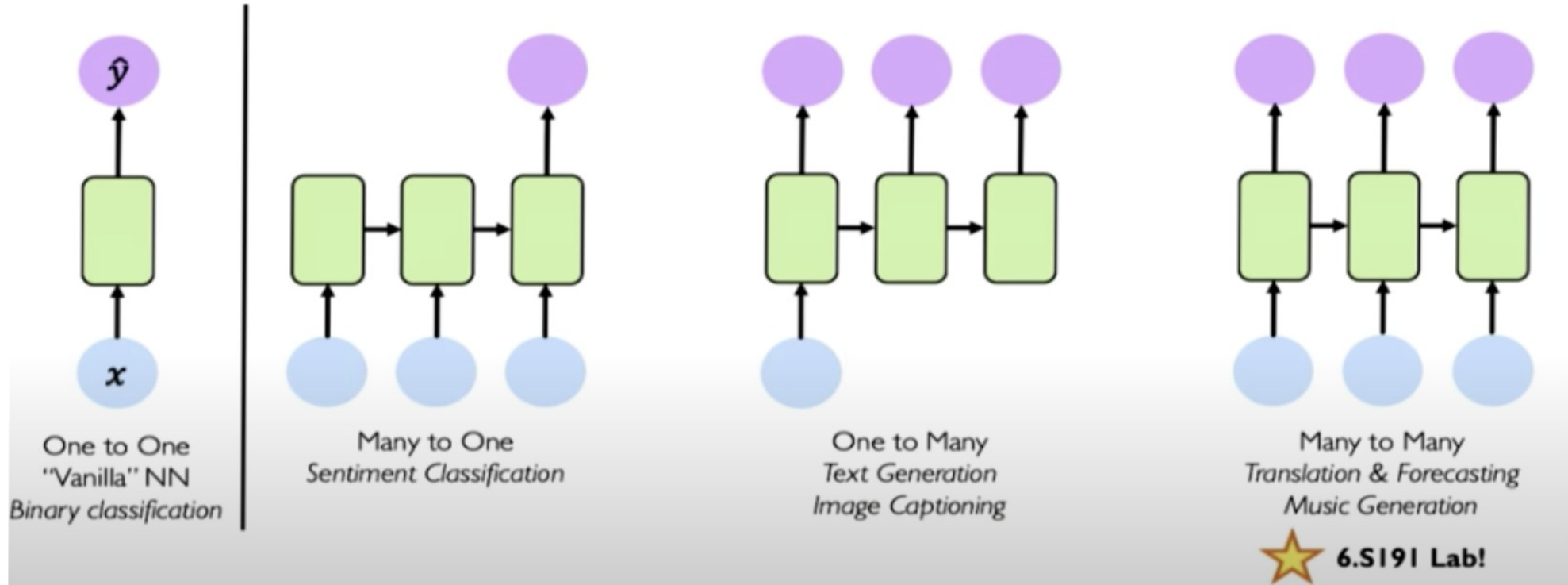
call function is the forward pass through.

1. update hidden state
2. compute the output
3. return output and hidden state

Use Tensorflow

`tf.keras.layers.SimpleRnn(rnn_units)`

RNNS for Sequence Modelling



Sequence Modeling : Design Criteria

- To model sequences we need to
 - Handle variable length sequences
 - Track long dependencies – Map something that appear early on to some that appear later in the sequence
 - Maintain information about the order
 - Share parameters across the sequence
- Recurrent Neural Networks (RNNs) meet these sequence modeling design criteria
-

A Sequence Modeling Problem

Predict the Next Word

- Problem : This morning I took my car for a walk
 - Given these words predict the next word
- How do you represent language to the neural network
- Embedding : Transform indices (Identifiers for objects) into a vector of fixed size
- How could this language embedding work for language data
 - Step 1 – Think about the overall vocabulary for your language (corpus) – What is the overall amount of words in your corpus
 - Step 2 – Match words to index (0..n-1)
 - Step 3 – Embedding – Index to fixed size vector
 - One Hot Embedding
 - Can make this a sparse vector (binary vector)
 - Learned Embedding – Use a machine learning model to learn an embedding of those words
 - Map the meaning of the words to an encoding that is more informative
 - more representative such that similar words (meaning) will have similar embeddings
 - Representative Learning – How we can take input and use neural networks to learn a meaningful encode for our problem of choice
- Handle variable length – Feed Forward Inputs cannot do this since they have inputs of fixed dimensionality. With RNNS your unrolling across time

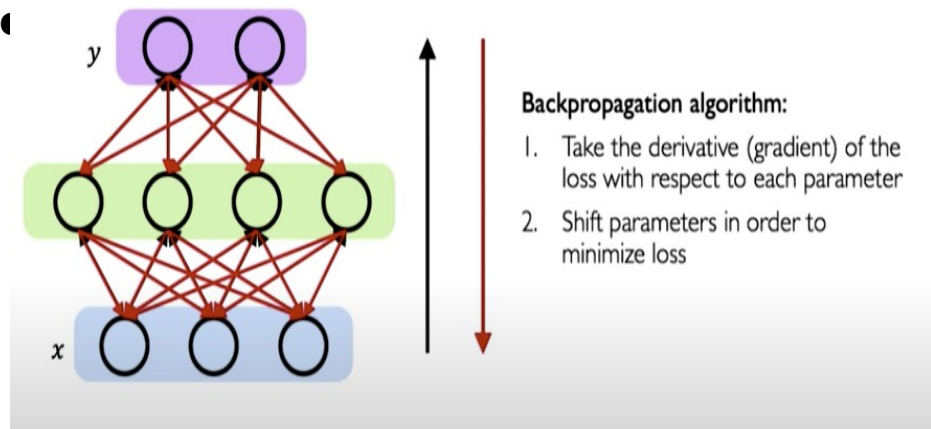
A Sequence Modeling Problem

Predict the Next Word

- Model long term dependencies – We need information from the distant past to accurately predict the correct word.
- Capture difference in the sequence order
 - Example : The food was good, not bad at all **and** The food was bad, not good at all
 - Have the exact same number of words and number of time they are used, but the meaning is different
- Share parameters across the sequence
- RNNs meet these sequence modeling design criteria
 - Handle variable length sequences
 - Track long dependencies – Map something that appear early on to some that appear later in the sequence
 - Maintain information about the order
 - Share parameters across the sequence
-
-

Back Propagation Through Time (BPTT)

- Need to review Back Propagation
- Use to train recurrent network models



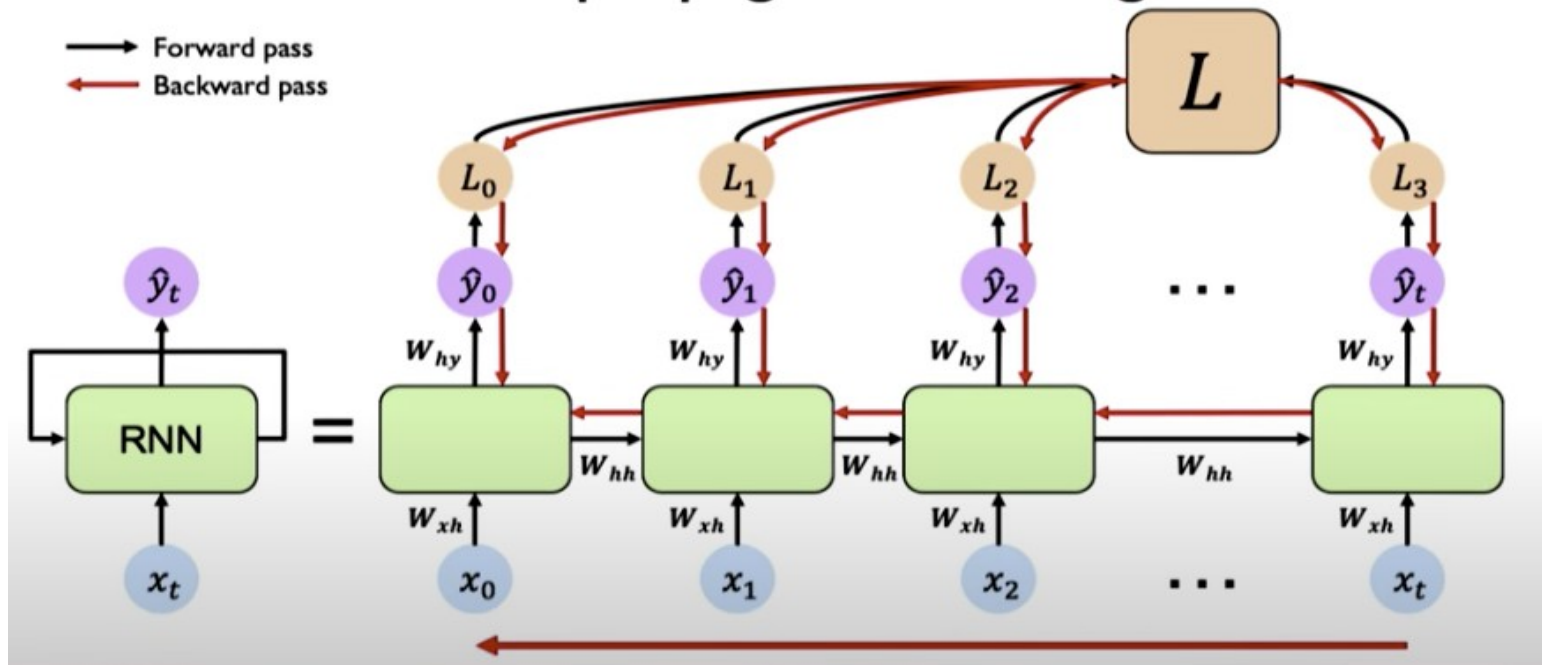
Go from input to output (forward) and back propagate our gradients downward through the network to adjust the weights

For RNNs we have to compute the loss values for each time step and then all time steps From where we currently are to the beginning of the sequence

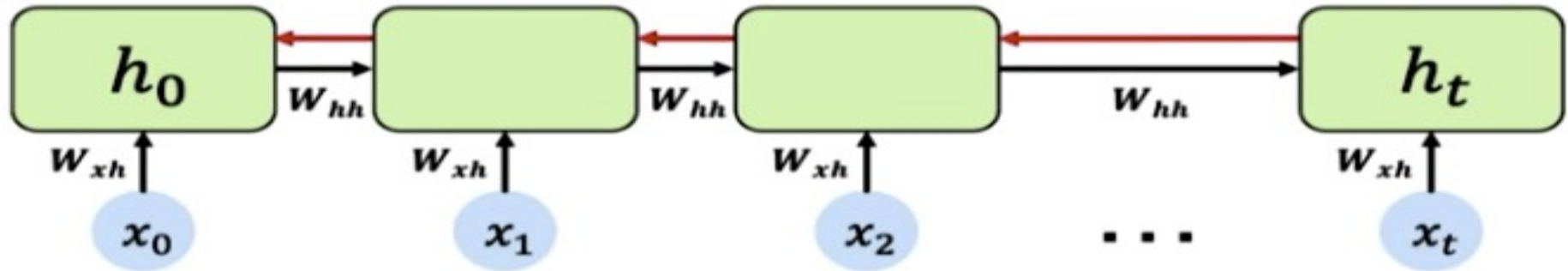
Back Propagation Through Time (BPTT)

- Back Propagate errors for their time step
 - From the current location to the start of the sequence (Errors flow backward in time)

RNNs: Backpropagation Through Time



Stadiant RNN Gradient Flow



Between each time step we need to perform individual matrix multiplications

Computing gradient wrt h_0 involves many factors of W_{hh} + repeated gradient computation.

Problems

Many values > 1 can cause exploding gradients and we cannot do optimization

Fix: Gradient Clipping to scale big gradients – Trimming the gradient values to scale back bigger gradients into a smaller value

Many values < 1 : Vanishing gradients and also a problem with training

Fixes:

- Activation Function

- Weight Initialization

- Network Architecture

Why are vanishing gradients a problem – Sabotage the goal of model long term dependencies

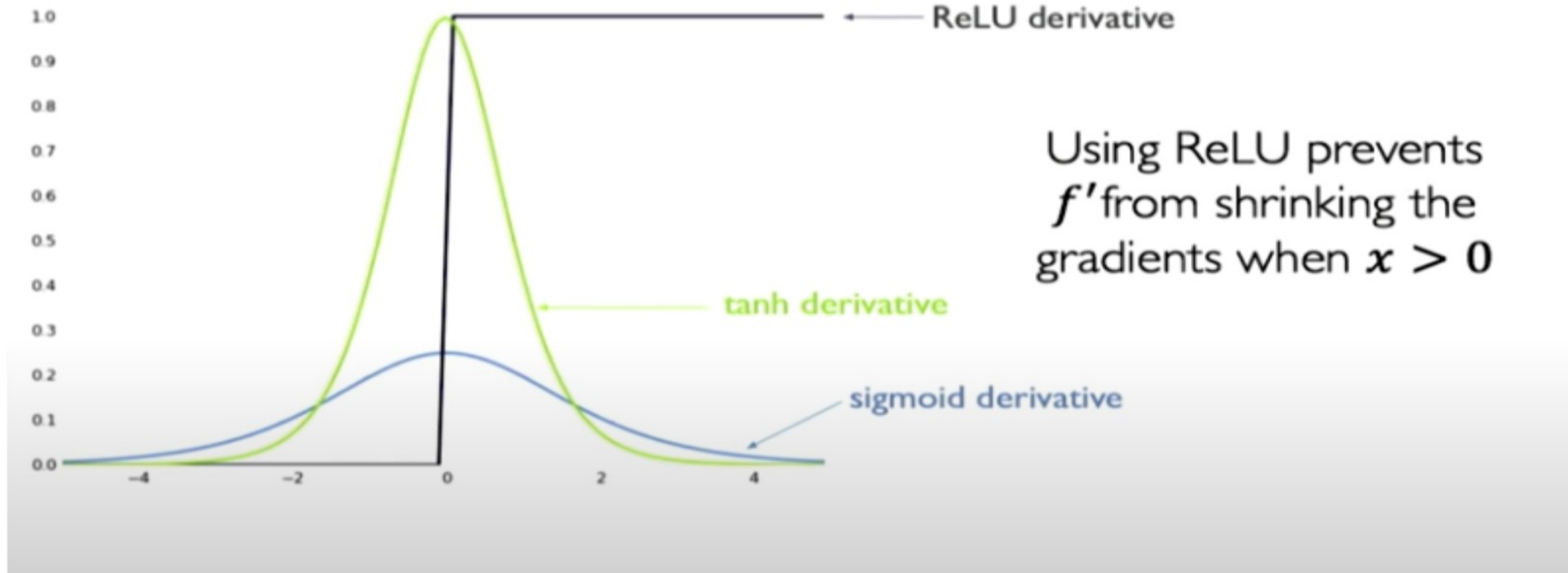
- Multiply many small numbers together bias the model to focus on Short Term Dependencies

 - Breaks down in longer sequences

- Errors due to further back time steps have smaller and smaller gradients,

- Bias parameter capture short term dependencies

Trick #1 : Activation Functions



Choose the activation gradient from shrinking dramatically – Relu

Trick #2 : Parameter Initialization

Initialize **weights** to identity matrix

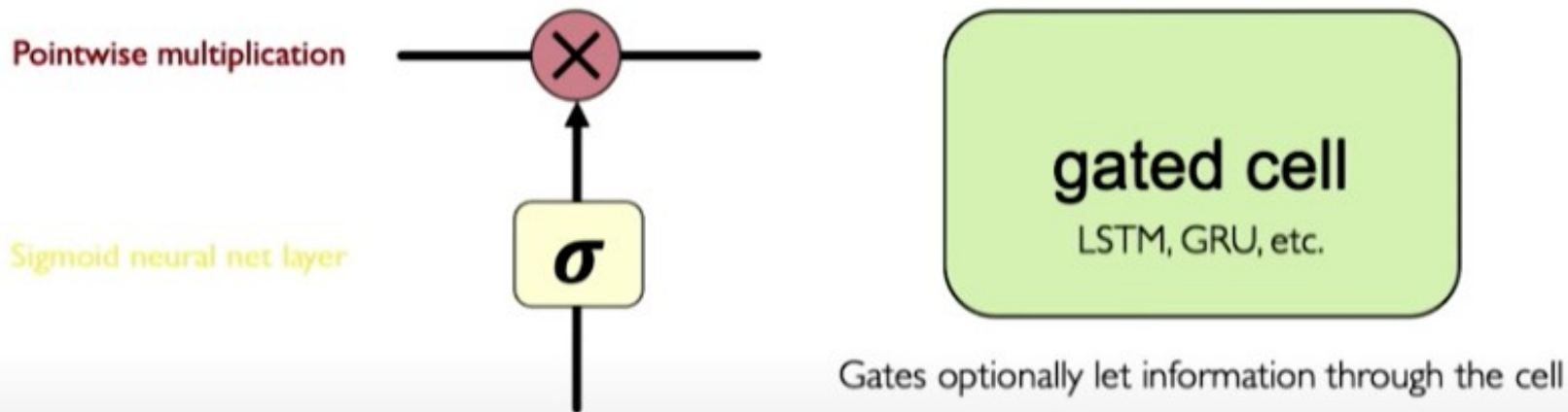
Initialize **biases** to zero

$$I_n = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}$$

This helps prevent the weights from shrinking to zero.

Trick #3 Gated Cells

Idea: use **gates** to selectively **add** or **remove** information within **each recurrent unit with**



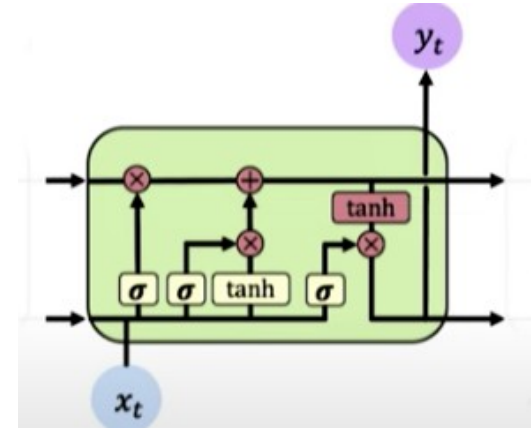
The most robust solution and uses gates to selectively add or remove information from the state within each recurrent unit (rnn).

The long term unit can track dependencies in the data

The gates can filter what data pass through the recurrence cell

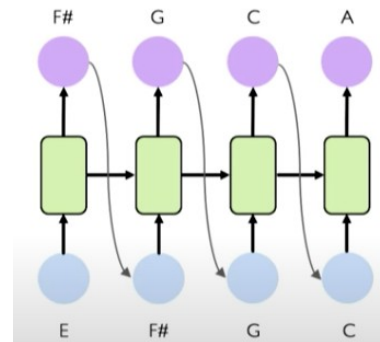
Long Short Term Memory (LSTMs) networks

- rely on a gated cell to track information through out many time steps.
- Gated LSTM cells control information flow:
 - Forget – Forget gate get rid of irrelevant information
 - Store – relevant information from current input
 - Update – Selectively update cell state
 - Output – gate returns a filtered version of the cell state
- LSTM cells are able to track information through many time steps
- `tf.keras.layers.LSTM(num_units)`
- The gates activate with each other to try and control information flow.
- Key Concepts
 - Maintain a cell state
 - Use gates to control the flow information for (forget, store, update, output)
- Backpropogation through time with partially uninterrupted gradient flow become much more stable and mitigate against the vanishing gradient problem by having fewer repeated matrix multiplications that allow fo for a smooth flow of gradients across out model.



Examples

- Music Generations – Input : Sheet music output : next character in sheet music
 - Use to generate brand new musical sequences
 - treat as next time step prediction problem
 - The output at each time step is the most likely note in the sequence
- Sentiment Classification – Input sequence of words , probability of having positive sentiment
 - `loss = tf.nn.softmax_cross_entropy_with_logits(y, predicted)`
 - Use case : Tweet sentiment classification – train on a bunch of tweets



Limitations of RNNs

- Encoding Bottleneck – Take a lot of content and condense it in a representation that can be predicted on.
 - Information can be loss in the encoding operation
- Slow, No Parallelization
 - Inefficient on the hardware
- Not long memory
 - Don't have high memory capacity, don scale well to sequences of length 1000s

Goals of Sequence Modelling

- trying to generate features to generate outputs
- With RNNs ; recurrence relation to model sequence dependencies
-

RNNs: recurrence to model sequence dependencies

Limitations of RNNs



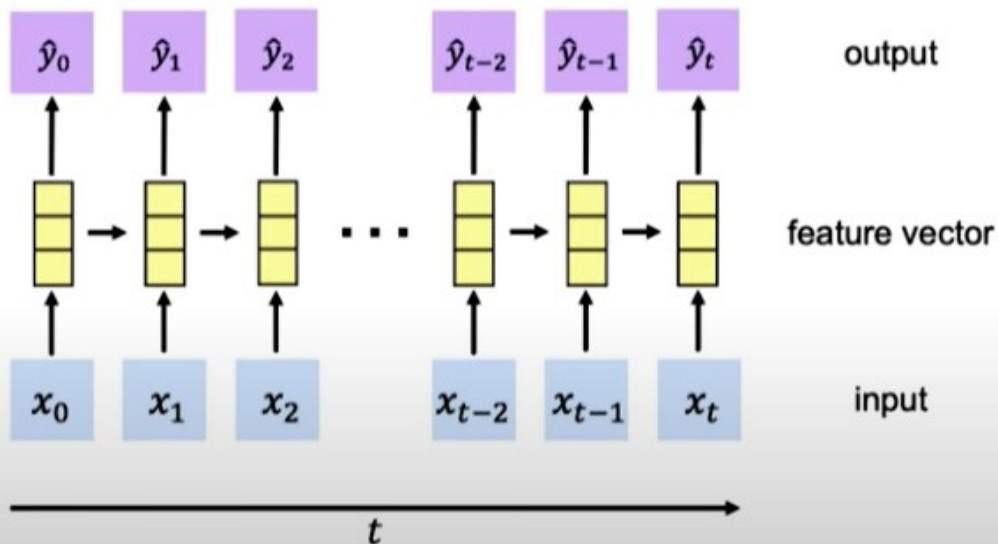
Encoding bottleneck



Slow, no parallelization

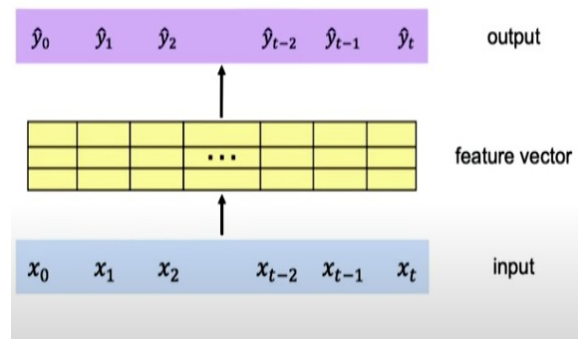


Not long memory



Goals of Sequence Modelling

- Desired capabilities instead of limitations
 - Continuous Stream instead of encoding issues
 - parallelization
 - long memory
- The limitations are caused since they process the time steps individually due to the recurrence relation
- Can we eliminate the need for recurrence entirely
 - We could squash the sequence together concatenating all the time steps into one vector with all the time steps then feed it into model, calculate the feature vector and feed it into a model
 - First approach (naive) – Take the squashed input and pass it to a fully connected framework
 - There will be no recurrence, but not scalable, no order and no long memory
 - We don't have a notion of what points are important
 -



Intuition Behind Self Attention

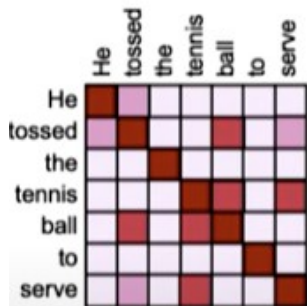
- Attending to the most important parts of input
- Given a picture of iron man how do we figure out what is important
 - naive – scan across the image what is important
 - our brains can easily pick up what is important and extract features
 - Parts of the problem
 - Identify which parts to attend to – Similar to a search problem
 - After doing a search (such as google) and for every item we get the key (information extracted)
 - Overlaps of the key and the query – (Compute Attention Mask) How similar is the key and query
 - Extract the features with high attention – Return the values with highest attention
 - Attention Mask – How similar each of the keys is to our query
- Concept of search is related to how self attention works in neural networks such as transformers

Learning Self Attention with Neural Networks

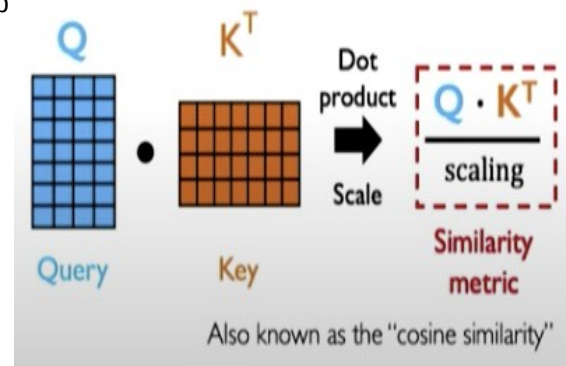
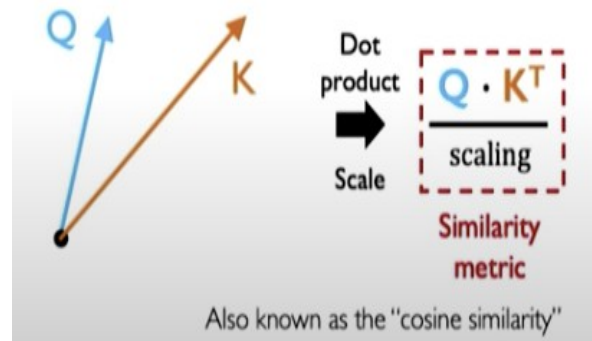
- Goal : Identify and attend to most important feature in the input without any need for processing the information time step by time step
- Steps
 - Encode position information
 - Data is fed in all at once! Need to encode position information to understand order
 - Create an embedding that incorporates some notion of position and combine them together
 - Extract query, key and value for search
 - Trying to learn a mechanism for self attention
 - operate on the input itself and only the input
 - create 3 new transformations that focus on the query, key and search – linear layers are different
 - For the query we take the positional embedding and linear layer and using matrix multiplication we create the Query
 - For the query we take the positional embedding and a different linear layer and using matrix multiplication we create a Key Transformer
 - For the query we take the positional embedding and a different linear layer and using matrix multiplication we create a value Transformer
 - Key, Query, Values are features
 - Compute attention weighting
 - Attention score : Compute pairwise similarity between the query and the key
 - How to compute similarity between two sets of features
 - the query and key are matrices or vectors how do we compute the overlap

Learning Self Attention with Neural Networks

- We have two vectors Q and K and compute the similarity by using the dot product and scaling
 - Can apply the same exact operation to metrics
 - Known as the cosine similarity
- Visualize the result
 - Use Softmax so that every value falls between a zero and one.
 - The metrics reflect a relationship between the components of the input to each other
 - This is a heat map visualization where the words related to each other
 - have a higher weighting (higher weighting)
 - The matrix is called Attention Weighting and captures the components of our relationship and to each other.
 - The entries relate the relation between the words. The higher the weighting the more related they are



$$\text{softmax} \left(\frac{Q \cdot K^T}{\text{scaling}} \right)$$



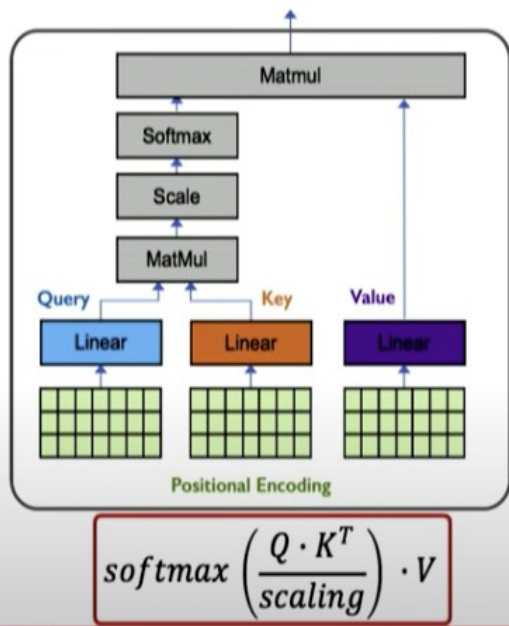
Learning Self Attention with Neural Networks

- Extract features with high attention
 - See the picture
 - reflects the features that correspond to high attention
- Summary – Copy the encoding information 3 times

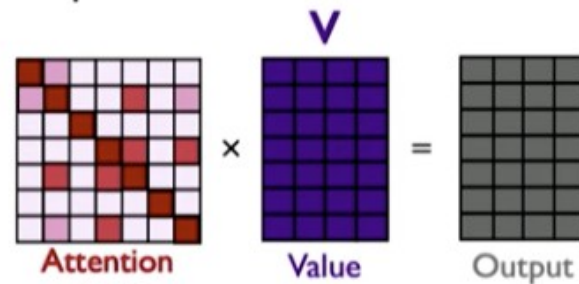
Goal: identify and attend to most important features in input.

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract **features with high attention**

These operations form a self-attention head that can plug into a larger network. Each head attends to a different part of input.



Last step: self-attend to extract features



$$\text{softmax}\left(\frac{Q \cdot K^T}{\text{scaling}}\right) \cdot V = A(Q, K, V)$$

Can have multiple self attention heads that concentrate on other features of the data

Language processing – Bert, GPT-3 - Create images based on sentences

Biological Sequences – AlphaFold 2 protein structure prediction

Computer Vision – Vision Transformers

Can do this multiple time. Have multiple head that pay attention to different things

Deep Learning for Sequence Modeling : Summary

- Summary
 - RNN are well suited for sequence modeling tasks
 - Model sequences via a recurrence relation
 - Training RNNs with back propagation through time
 - Models for music generation, classification, machine translation and more
 - self attention to model sequences without recurrence

Self-Attention Applied

Language Processing

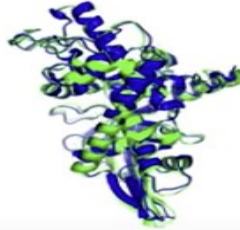


An armchair in the shape
of an avocado

BERT, GPT-3

Devlin et al., *NAACL* 2019
Brown et al., *NeurIPS* 2020

Biological Sequences



AlphaFold2

Jumper et al., *Nature* 2021

Computer Vision



Vision Transformers

Dosovitskiy et al., *ICLR* 2020

Deep Computer Vision

Convolutional Neural Networks

- Vision – To discover from images what is present in the world, where things are and what actions are taking place to predict and anticipate events in the world
 - Accounting for all details in a scene
- The impact of computer vision
 - Robots
 - Mobile Computing Applications
 - Diagnose Driving
 - Autonomous driving
- Deep Learning can learn from raw pixels and extract meaningful features
 - Facial detection and recognition
 - Self Driving Cars
 - Medicine and Biology
 - Accessibility to help the digitally impaired (detect trails for running

What Computer See

- What Computers See
 - To a computer images are numbers
 - can represent them as a matrix
- Now that we have a data structure
 - Regression – output variable takes continuous value
 - Quantitative Analysis
 - Continuous Vaue
 - Classification – Output variable takes class label. Can produce probability of belonging to a particular class
 - Recognition
 - A label and probability
 - Our pipeline needs to tell what is unique to the picture.
 - At a high level what features distinguish each of those images
 - If the features are present then there is good chance of a match
-

Manual Feature Extraction

- Domain Knowledge
- Define features
 - Done by humans, but not very good at describing all the variations (See pictures)
 - variations make it very hard to define the features what features the algorithm may need to identify
- Detect features to classify
 - Manual extraction will breakdown in the detection task
 - Using features defined by the human may breakdown in the detection task
 - The detection of features is problematic with variation
- Solutions
 - Detect an feature automatically by observing a bunch of features
- Learning feature representations
 - Can we learn a hierarchy of features directly from the data instead of hand engineering
 -



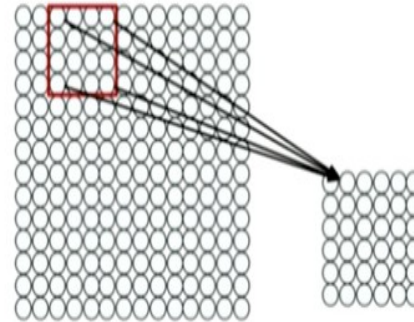
Learning Visual Features

- Which architecture should we use
- Fully Connected Network
 - Input : 2D Image, Vector of pixel values - we have flatten out all the data and lost the spatial structure
 - fully connected
 - Connect neuron in hidden layer to all neurons in input layer
 - No Spatial Information – pixels that were close to each other now may be very far apart
 - And many parameters
- How can we use spatial structure in the input to inform the architecture of the network
 - Use Spatial Structure
 - Input 2D Image – Array of pixel values
 - Idea : Connect patches of input to neurons in hidden layer
 - Neuron connected to region of input : Only sees the values
 - Feed the image data (subset to a patches of weights)
 - Each neuron will only see a subset of pixels at any give time
 - Feed the a subset of the pixels to a patch of weights
 - pixels that are close to each other share information – A plant in the pond (Since the pond is there you know it is a pond plant)

Feature Extraction with Convolution

- Using Spatial Structure to defined connections across the image
 - Connect patch in input layer to a single neuron in subsequent layer.
 - Use a sliding window to define connections
 - Each time it slides we will predict the next single neuron input
 - Same patch for all slides – want to use that feature that we learned and use it all across the image
 - By sliding it time over the image we can create an extraction of features which is another 2D Matrix
 - We weigh the connection between the patch and the neurons that are the output
 - How can weight the patch to detect a particular features

- Feature Extraction
 - Apply a set of weight – a filter to extract local features
 - Use multiple filters to extract different features
 - Spatially share parameters of each filter
- Give a set of features (a small weight matrix) how can we extract those features



- Filter of size 4x4 : 16 different weights
- Apply this same filter to 4x4 patches in input
- Shift by 2 pixels for next patch

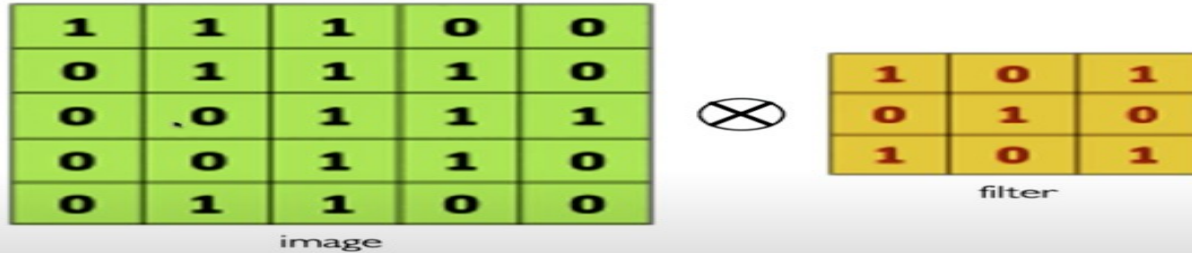
This “patchy” operation is **convolution**

the result of the filter is the state of the neuron in the next layer

Feature Extraction and Convolution

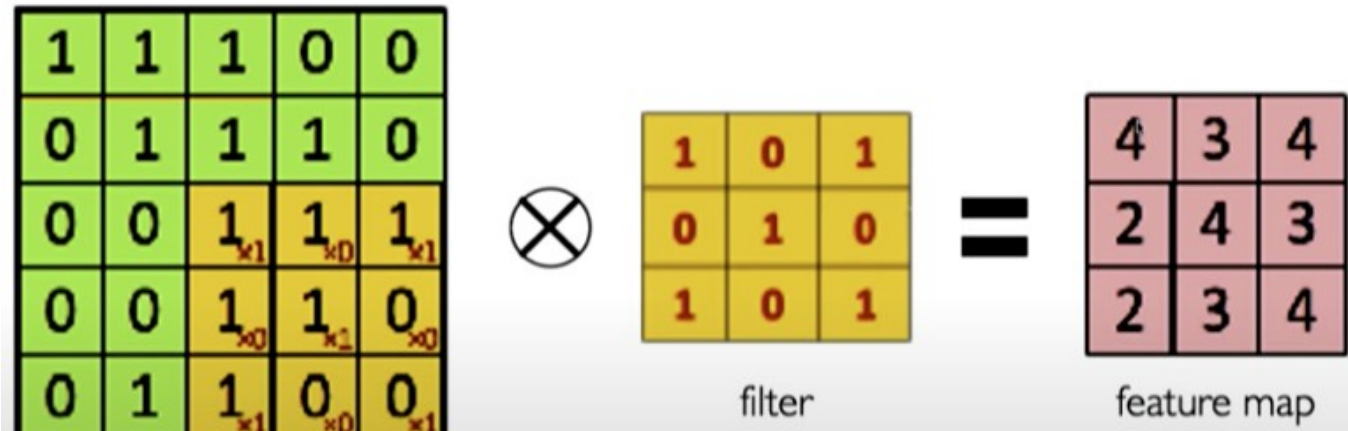
A Case Study (2) – The Convolution Operation

Suppose we want to compute the convolution of a 5x5 image and a 3x3 filter:



We slide the 3x3 filter over the input image, element-wise multiply, and add the outputs...

We have a feature map the strength at every single location detecting that feature at. Where there is a large number there is a large result. We can see where the feature was detected

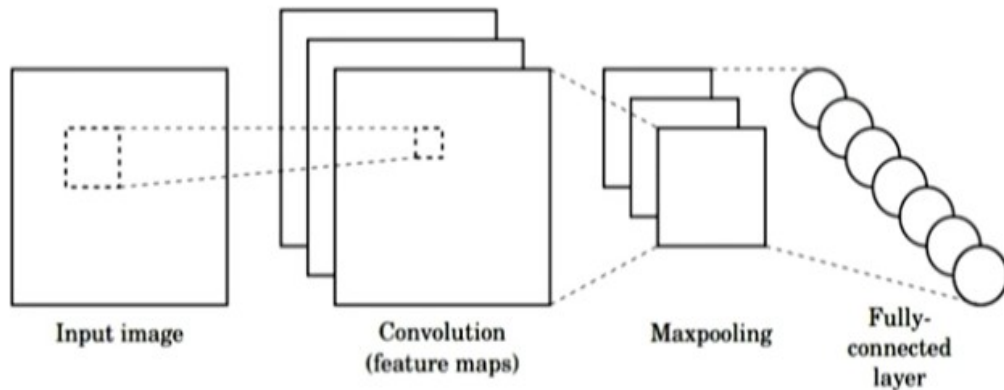


Producing Features Maps

- Different Feature Maps – Produced by different filters (features)
 - sharpen
 - Edge Detect
 - Strong Edge Detect
- By changing the weights in our filter out model is able to learn different types of features
- Convolution – Capitalize on spatial structure and use a set of weight to detect features
-

Convolutional Neural Networks (CNN)

- Convolution : Apply filters to generate feature maps
- Non-Linearity : Often ReLU
- Pooling : Downsampling operation on each feature map. Our filter can attend to a larger region of input space
- Train the model with image data. Learn weights of filters in convolutional layers.
- Tensor flow commands
 - `tf.keras.layers.Conv2D`
 - `tr.keras.activations.*`
 - `tf.keras.layers.MaxPool2D`
- Goal learn the features from the image data and use the features to identify the properties in the image and classify it



Convolutional Layers – Local Connectivity

- For a neuron in hidden layer:
 - Take inputs from patch
 - Compare weighted sum
 - Apply bias

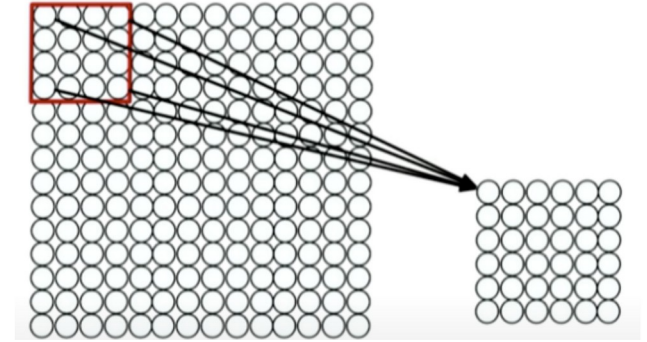
- Mathematical Representation

- Apply a window of weights
- computing linear combinations
- activating with non line function

4x4 filter: matrix
of weights w_{ij}


$$\sum_{i=1}^4 \sum_{j=1}^4 w_{ij} x_{i+p, j+q} + b$$

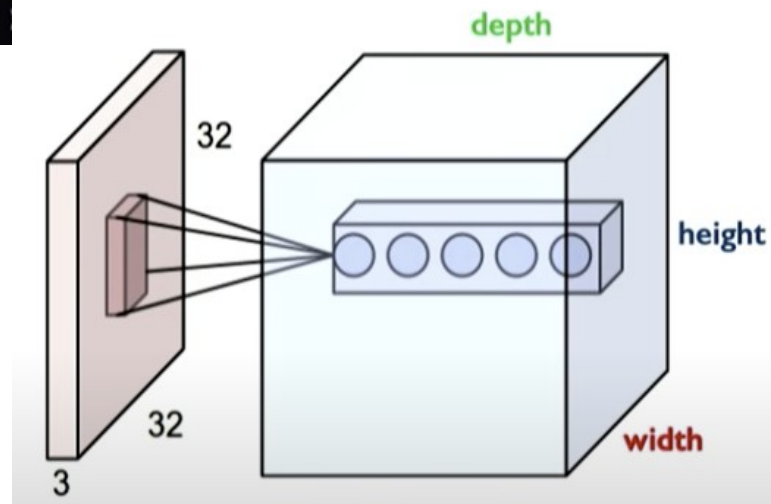
for neuron (p,q) in hidden layer



- Define how neurons in one layer (input layer) are connected to neurons in the output layer

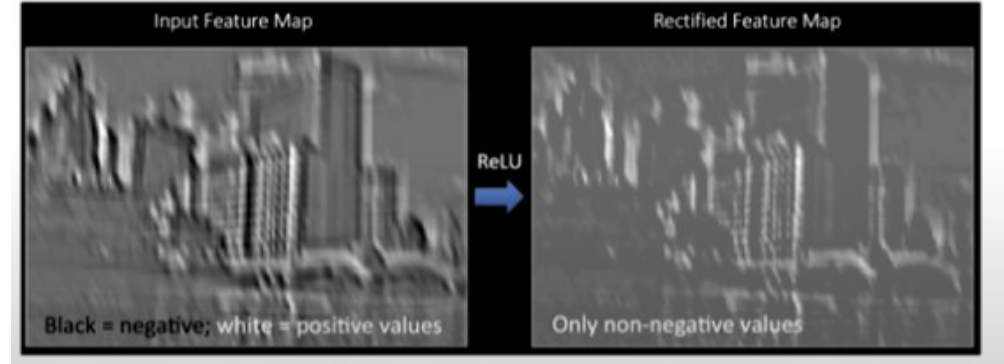
CNNs: Spatial Arrangement of Output Volume

- Tensorflow –  `tf.keras.layers.Conv2D(filters=d, kernel_size=(h,w), strides=s)`
- Layer Dimensions
 - $h \times w \times d$
 - where h and w are spatial dimensions
 - d (depth) is number of filters
- Stride : Filter Step Size
- Receptive Field : Locations in input image that a node path is connected to.
- In a single convolutional layer that can have multiple filters.
 - The output of each later is a volume of images. One image for each filter learned
 - The locations The parameters define some spatial arrangement of the output

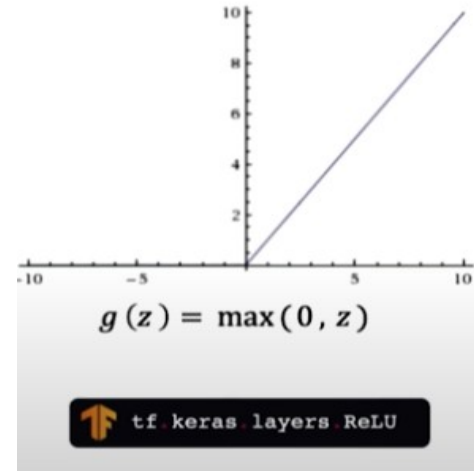


Introducing Non Linearity

- After every convolution operation (I.e after convolutional layers)
- ReLU pixel by pixel operation that replaces all negative values by 0.
 - Non-linear Operation
- The images are non linear so we need to apply Non Linearities
- ReLU – An identify function when your posive
 - $x < 0$ does not get pass through
 - positive number – positive detection of feature
 - 0 – No detection of the feature
 - negative number inverse detection

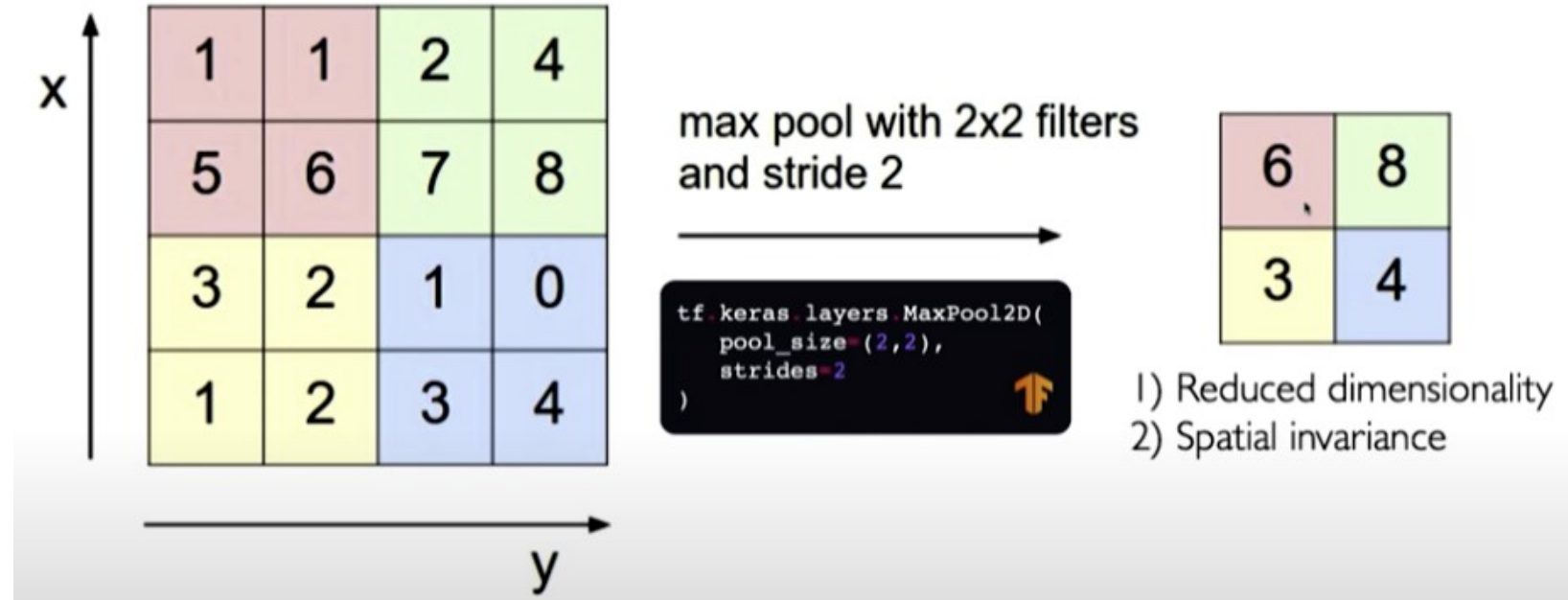


Rectified Linear Unit (ReLU)



Pooling

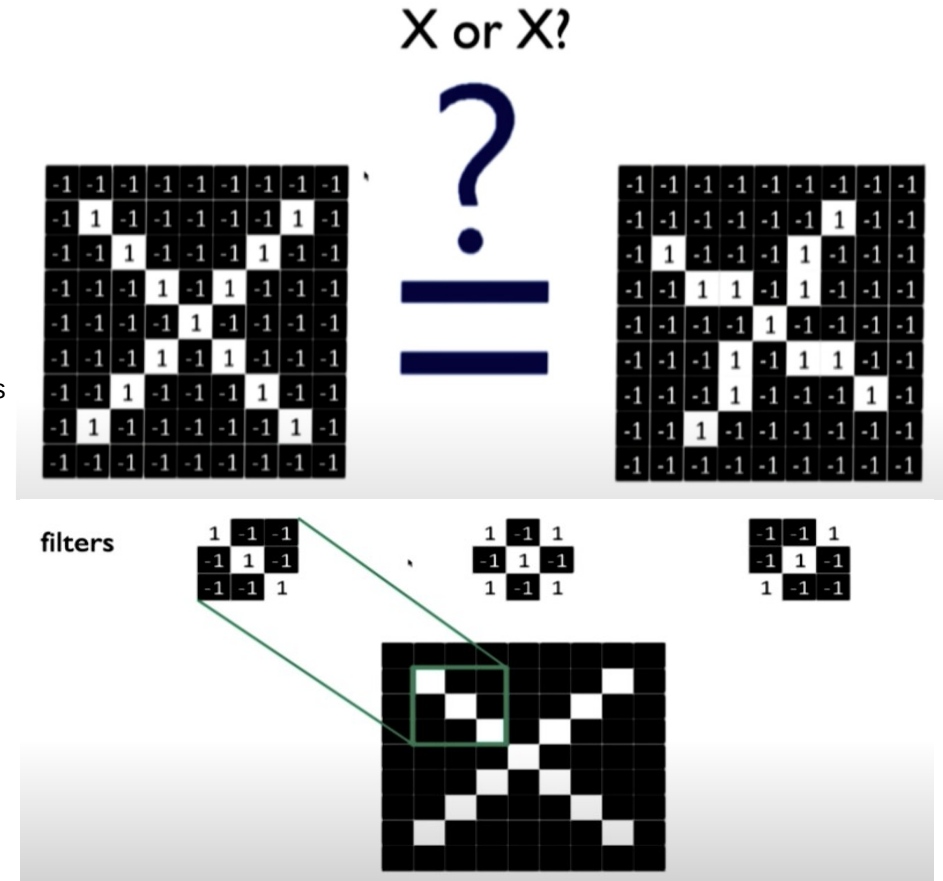
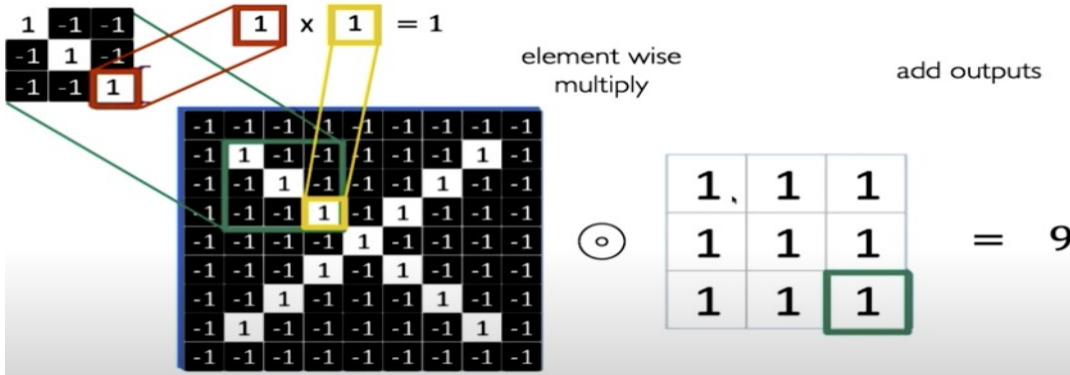
- Reduce the dimensionality and make the model scalable which still preserving spatial variance and structure
- Max Pooling – Select Patches by taking the max of each of the patches



Feature Extraction and Convolution

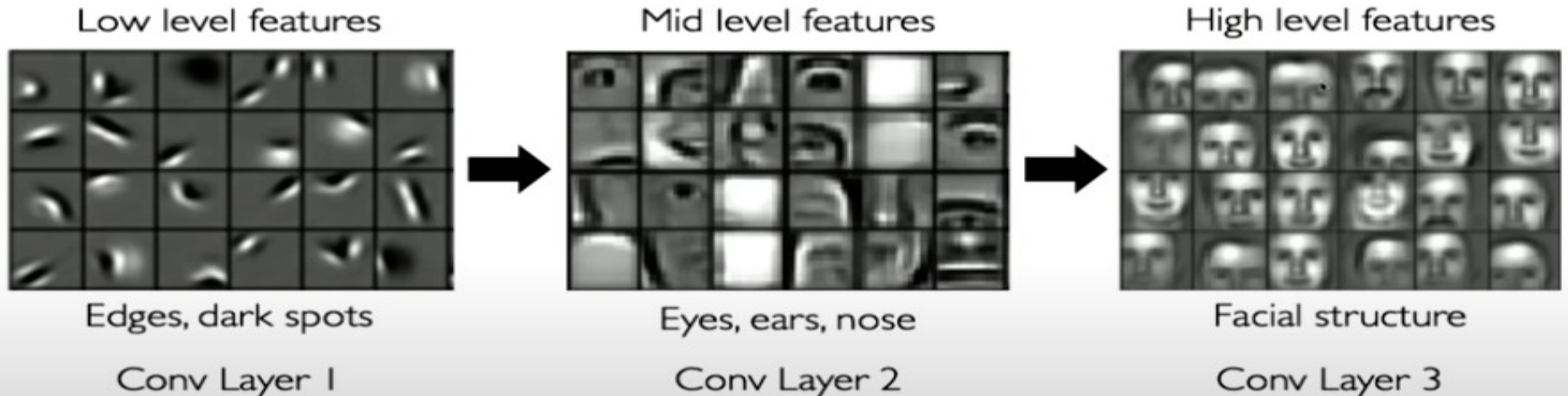
A Case Study

- Image is represented as matrix of pixel values... and computers are literal
 - We want to classify X even if it is shifted/rotated/deformed.
- compare patch by patch and identify certain features and detect those
- Each feature is mini image : A small matrix of values
- Define an operation that will connect the filter to the image
 - convolution – Takes two images and outputs a third image
 - preserve relationship of pixels by learning image features in small squares
 - The nine is to be the output of the convolution at this location



Representation Learning Deep Cnn

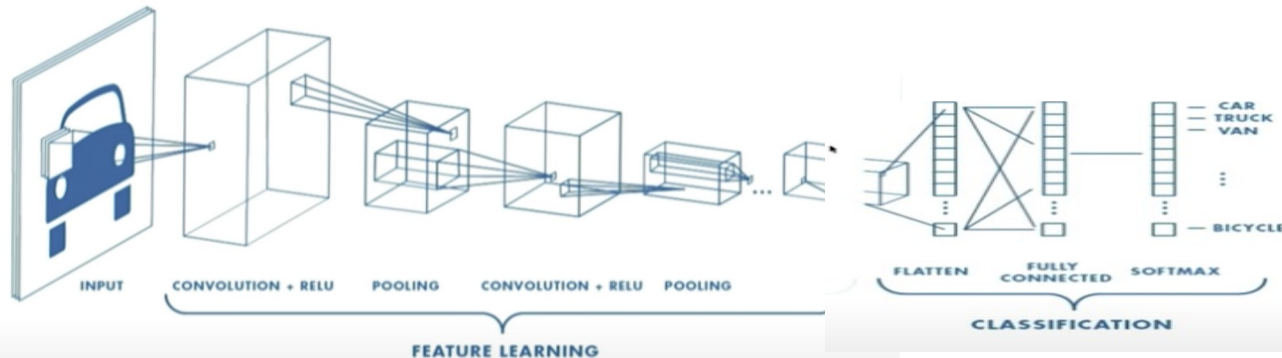
- learn the high level features from one convolutional layers to the next layer



CNNs for Classification: Feature Learning

- Stack these three steps in the form of a neural network
 - 'learn features in input through convolution – could be a set of filter which produce an feature volume
 - Introduce non-linearity through activation function (real-world data is non linear)
 - Reduce dimensionality and preserve spatial invariance with pooling
- Keep Stacking the layers over and over again and get a set of feature volumes that we can take those features which now consume a small amount of memory and feed them through a fully connected layer for the decision making process
- CONV and Pool layers output high level features of input
- Fully Connected layer uses these features for classifying input image
- Express output as probability of image belonging to a particular class

$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$



Code

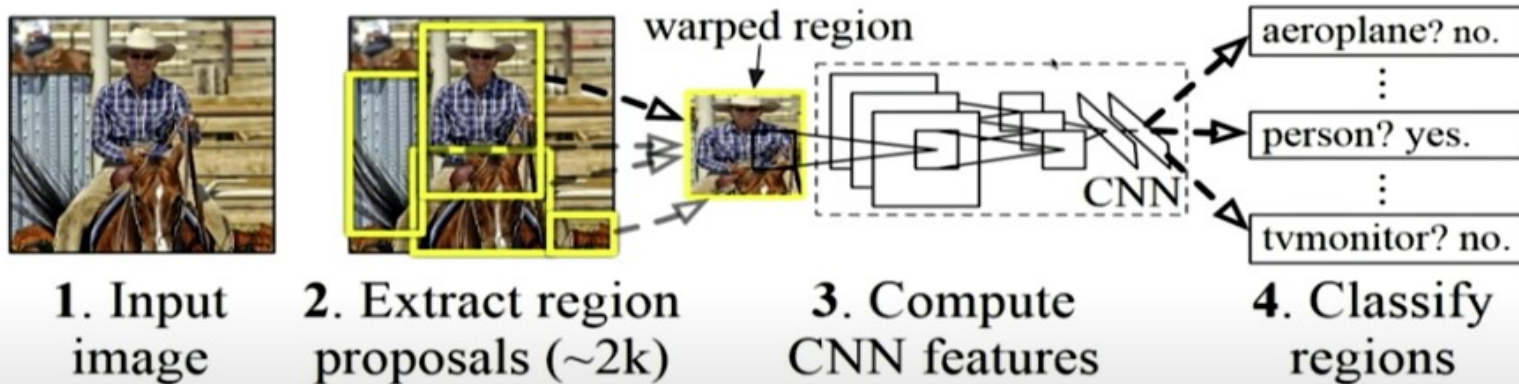
- `import tensorflow as tf`
- `def generate_model()`
 - `# First Convolutional layer – Learn 32 different type of features`
 - `model = tf.keras.Sequential([`
 - `tf.keras.layers.Conv2D(32, filter_size=3, activation='relu'),`
 - `tf.keras.layers.MaxPool2d(pool_size=2, strides = 2)` `// Down Scale`
 - `# Second Convolutional layer – Learn 64 features since we have downscaled`
 - `tf.keras.layers.Conv2D(64, filter_size=3, activation='relu'),`
 - `tf.keras.layers.MaxPool2d(pool_size=2, strides = 2)`
 - `# Fully Connected Classifier – flatten into a set of features.`
 - `tf.keras.layers.Flatten()`
 - `tf.keras.layers.Dense(1024, activation='relu'),`
 - `tf.keras.layers.Dense(10, activation='softmax')`
 - `])`

An Architecture for Many Applications

- Two parts of the architecture
 - First part focuses on Feature Extraction
 - Second Part = Classification, Object Detection, Segmentation, Probabilistic Control
- Classification : Breast Cancer Screening
 - CNN Based system outperformed expert radiologists at detecting breast cancer from mammograms
 - From a single image we want to list of bounding boxes (Object detection)
- Object Detection
 - Image X sent through a CNN → Gets labelled as a Taxi
 - A harder problem of putting a bounding box for each image
 - very hard due to the large number of objects
 - The network must be very flexible and infer many dynamic images
 - How to implement
 - Naive
 - Place a random white box somewhere and feed it through the CNN and keep trying boxes until we find an image
 - Problem : Way too many inputs! This results in too many scales, positions and sizes

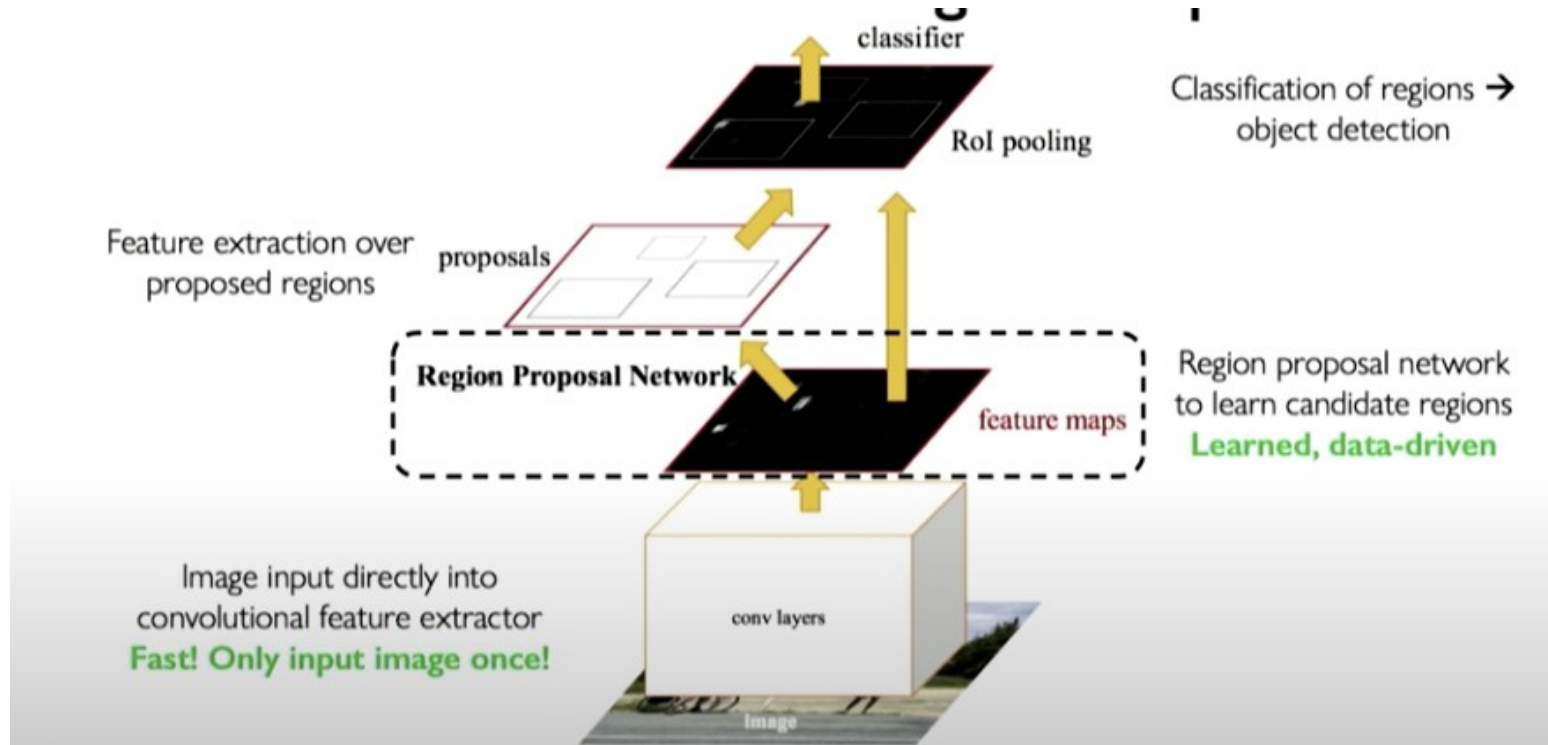
An Architecture for Many Applications

- R-CNN algorithm find regions that we think have objects. Use CNN to classify
- Instead of picking random boxes use a heuristic to pick the boxes.
 - The heuristic looks for a box with some signal in the image.
- The algorithm will shrink the box down
- Still slow since we have to feed each region that heuristic and check is there a class or not
- Extremely brittle since the feature extraction and extracting the regions are totally separate
 - They should be very related extract boxes where we see features
- Summary Problems : Slow! Many regins; time intensive inference
 - Slow Many regiiouns time intensive inference
 - Brittle! Manually defined region proposals
 -



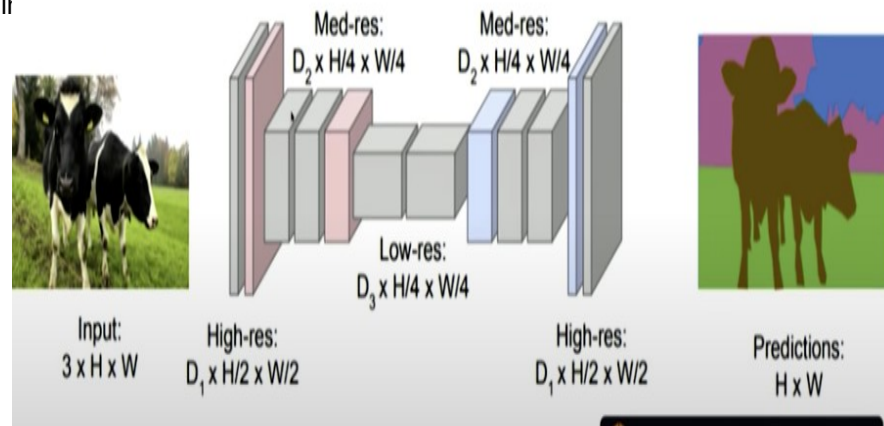
An Architecture for Many Applications


- Variations – Faster R-CNN Learns Region Proposals
 - Extremely Fast : The feature extraction model is learned with the downstream classifier



Semantic Segmentation : Fully Convolutional Networks

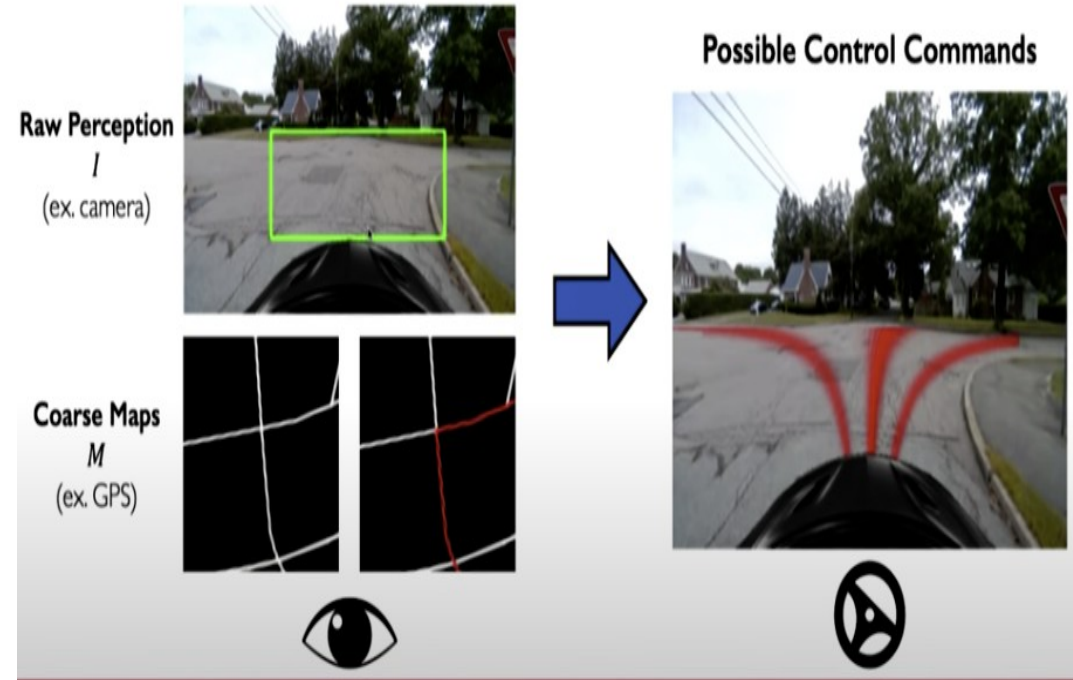
- Networks designed with all convolutional layers with downsampling and upsampling
- For every single pixel we want to predict what is the class for this pixel
 - super high input dimensional space
 - Here we are picking the cow pixels from the tree pixels from the grass pixels from the sky pixels
- Parts of the Solution
 - Feature Extraction Model Encoding
 - Upscaling Operation use Transpose Convolutions on the right
 - Output – Pixel Wise Classes
- Can be applied to biomedical images segment cancerous regions for the brain



 `tf.keras.layers.Conv2DTranspose`

Continuous Control : Navigation from Vision

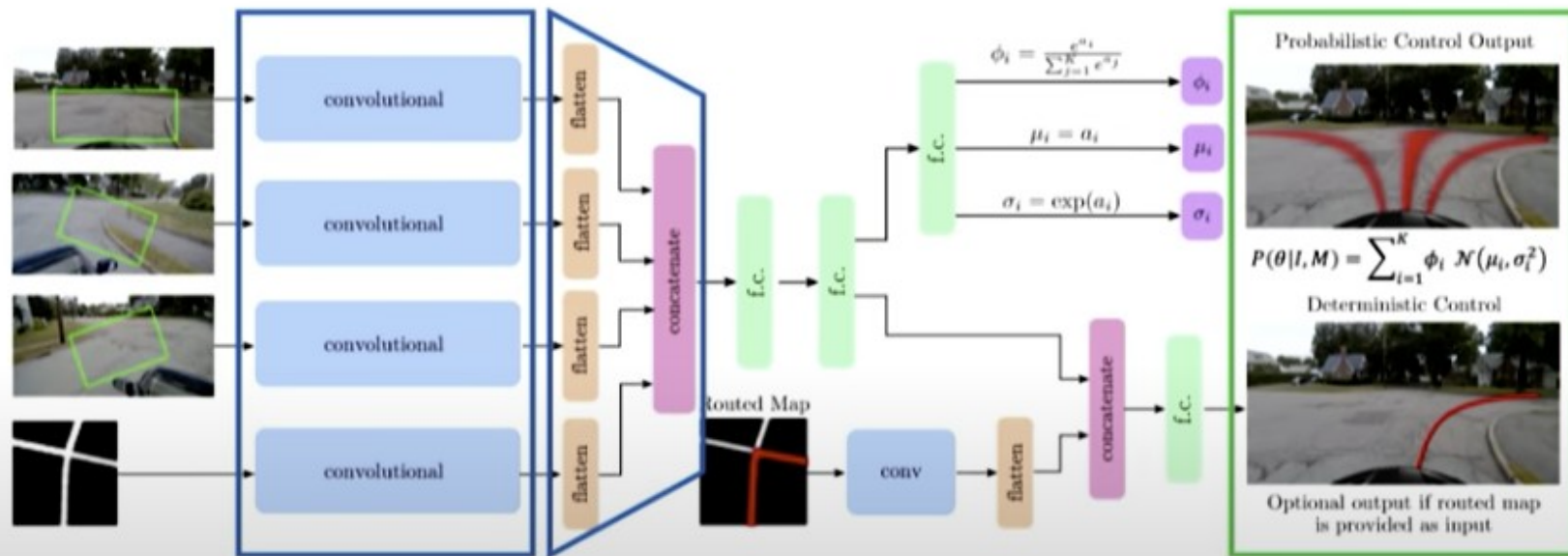
- A model from raw perception to some inference of the steering angle.
- Output a continuous distribution
- Notes on the solution
 - Entire mode is trained end to end without any human labelling or annotations
 - The Cameras are through their own feature extractor
 - The features are all combined together
 - Have a giant set of feature that is our entire environment
 - Loss function
 - The top part of the model is a fully connected layer which input the features and output the parameters of this continuous distribution
 - The Bottom enables learning the probability learning probability distributions even though the human never took all actions. It can learn to maximize that action in the feature. After seeing the intersections (computer) there is a key feature to permit me to turn all the different directions.



Continuous Control : Navigation from Vision

End-to-End Framework for Autonomous Navigation

Entire model is trained end-to-end **without any human labelling or annotations**



$$L = -\log(P(\theta|I, M))$$

Summary

Deep Learning for Computer Vision: Summary

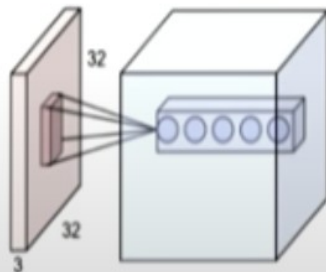
Foundations

- Why computer vision?
- Representing images
- Convolutions for feature extraction



CNNs

- CNN architecture
- Application to classification
- ImageNet



Applications

- Segmentation, image captioning, control
- Security, medicine, robotics



Deep Generative Modeling

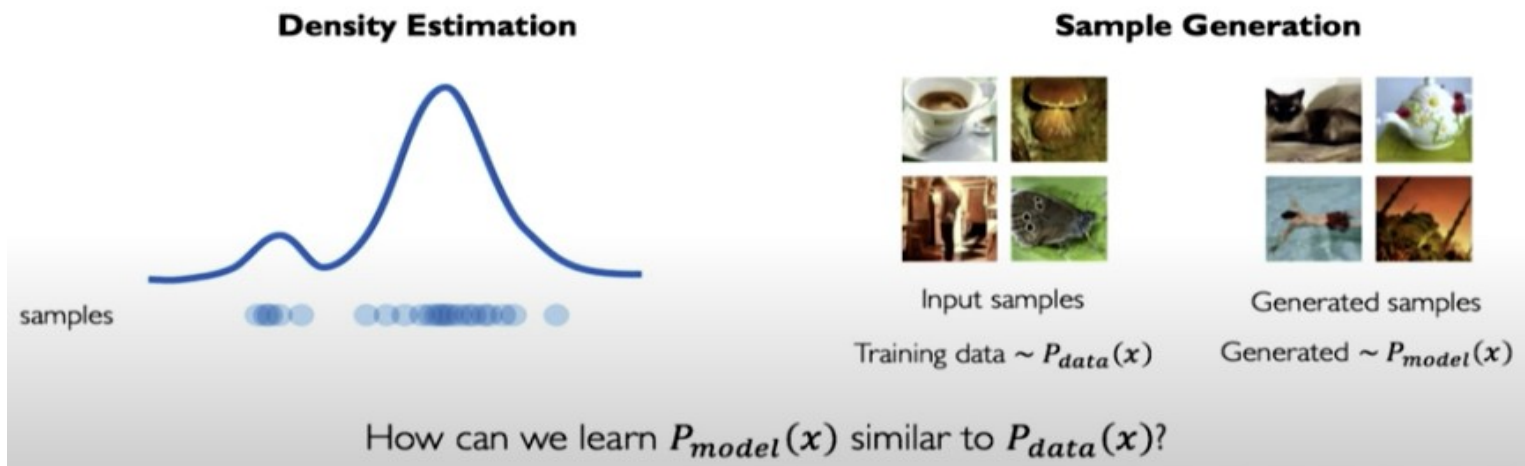
- Building systems that look for patterns in existing data and generate brand new data instances
- Generative Modeling – A field in deep learning

Supervised Learning vs Unsupervised Learning

- Supervised
 - Data (x,y)
 - X is the data and y is the label
 - Goal learn function to map x to y
 - Examples Classification, Regression, object detection, semantic segmentation
- Unsupervised
 - Data X is data no labels
 - Goal learn some hidden or underlying structure of the data
 - Examples: clustering, Feature or dimensionality reduction etc.

Generative Modeling

- Goal : Take as input training samples from some distribution and learn a model that represents that distribution
 - Can do this using
 - Density Estimation – Learn an approximation of what the probability distribution could be
 - Sample Generation – Learn a model; of the data distribution to generate new instances (samples)
 - The task is to learn a probability distribution of the data is very high dimensional.
 - Use neural networks to extraordinary complex functional mappings and estimates of these high dimensional data distributions



Why Generative Models

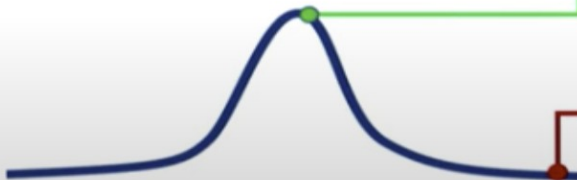
- Capable of uncovering underlying features in a dataset
- How can use use this information to create fair and representative datasets
- Because of the probability distribution they are uncovering what could be the underlying features in an unsupervised manner
- Want to understand what our distributions look like in a downstream task
- Outlier Detection

- **Problem:** How can we detect when we encounter something new or rare?
- **Strategy:** Leverage generative models, detect outliers in the distribution
- Use outliers during training to improve even more!

95% of Driving Data:
(1) sunny, (2) highway, (3) straight road



Detect outliers to avoid unpredictable behavior when training



Edge Cases



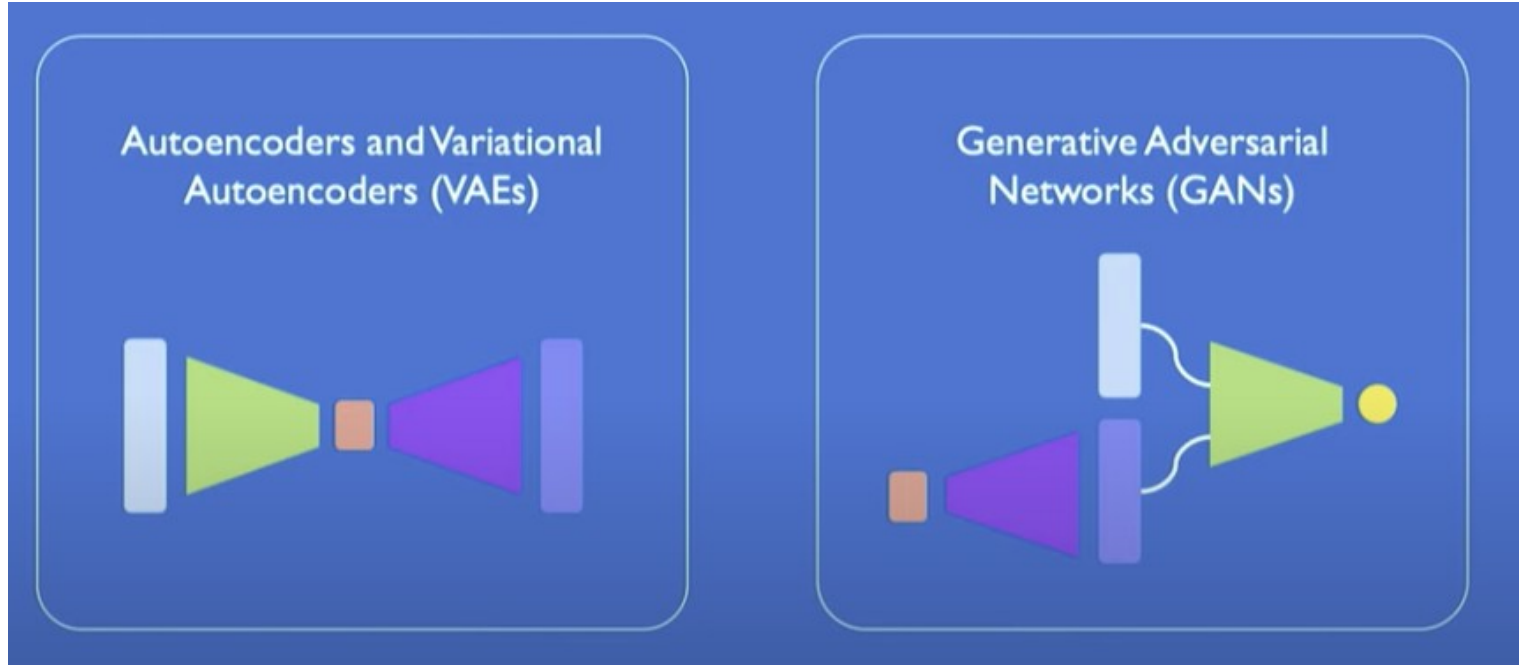
Harsh Weather



Pedestrians

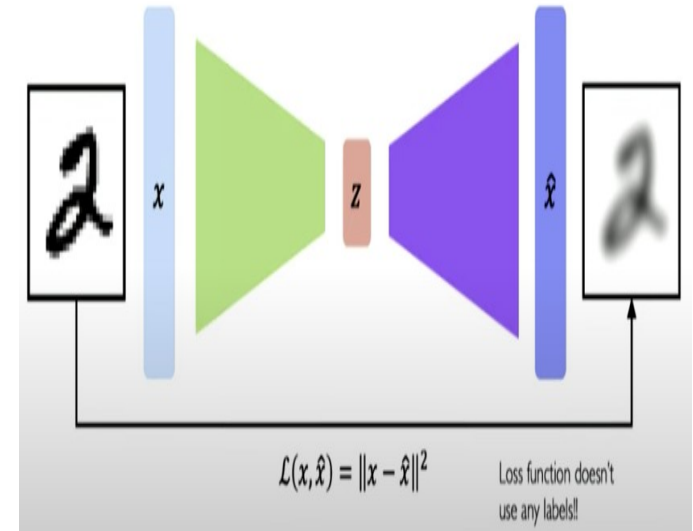
Latent Variable Models

- Latent Variable – Underlying variables that are governing some behavior, but we cannot directly observe
- Generative Modeling – To find ways to actually learn what the latent variables can be using only the observed data
-



AutoEncoders : Background

- A simple generative model is to build some encoding of the input and try to reconstruct an input directly
- Unsupervised approach for learning lower dimensional feature representing from unlabeled training data
- Learns mapping from the data x to a low dimensional latent space Z
 - Input : Raw Data and passthrough Deep Neural layers and the output space is low dimensional latent space
 - Why do we care about low dimensional Z
 - build a compression of the data
 - get a compact and meaningful representation
- If the goal is to predict vector Z we don't have labels for Z they are underline and hidden
 - How can we train a network
 - Solution : Train the model data to use these features to reconstruct the original data
 - Decoder learns mapping back from latent space z . to a reconstructed observation \hat{x}
 - The reconstructed image is an imperfect reconstruction
- Train the network – (Mean Square Error) $\mathcal{L}(x, \hat{x}) = ||x - \hat{x}||^2$
 - pixel by pixel difference
 - Use no lables



Dimensionality of latent space → reconstruction quality

Autoencoders for Representation Learning

- Autoencoding is a form of compression
 - Smaller latent space will force a larger training bottleneck
 - Compression – lowering of our dimensionality
 - Degree to how we perform the compression has a big affect o0n how good our reconstructions turn out to be.
 - The few latent variables we try to learn the worse our quality
- Autoencoders for representation learning
 - Representation Learning – Trying to learn a compressed representation of our input data without any sort of label
 - Building an automatic encoding of the data
 - self encoding the input data
 - Bottleneck hidden layer – force network to learn a compressed latent representation
 - Reconstruction Loss forces the latent representation to capture (or encode) as much information about the data as possib
 - Autoencoding = Automatically encoding data
 - Auto = self-encoding

VAE Optimization

Priors on the latent distribution

- regularization term
 - represent a probability distribution between between
 - computation our encoder is trying to learn Encoder computes: $q_{\phi}(z|x)$
 - Inferred latent distribution || fixed prior on latent distribution
 - prior $p(z)$ – some initial hypothesis what the latent variable z could look like
 - help the network enforce some structure based on the prior such that the learn latent variables follow the prior distribution
- What D is trying to – prevent the network from overfitting on certain restrictive parts and going to wild
 - minimize the distance between the latent distribution and the notion of a prior

Inferred latent distribution

$$D(q_{\phi}(z|x) \parallel p(z))$$

- Priors on the latent distribution



Common choice of prior – Normal Gaussian:

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

- Encourages encodings to distribute encodings evenly around the center of the latent space
- Penalize the network when it tries to “cheat” by clustering points in specific regions (i.e., by memorizing the data)

$$D(q_{\phi}(z|x) \parallel p(z))$$

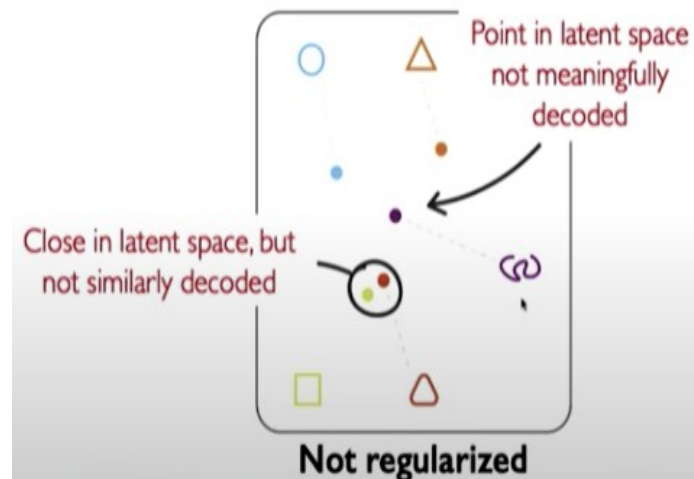
Inferred latent distribution Fixed prior on latent distribution

$$= -\frac{1}{2} \sum_{j=0}^{k-1} (\sigma_j + \mu_j^2 - 1 - \log \sigma_j)$$

KL-Divergence

Intuition on regularization and the Normal Prior

- What properties do we want to achieve from regularization
 - Continuity : points that are close in latent space → Similar content after decoding
 - Completeness: sampling from latent space → Meaningful content after decoding
 - points close in latent space and meaningfully decoded
- How can the normal prior help use achieve this regularization
 - Encoding latent variables according to a non regularize does not guarantee continuity and completeness
 - small variances could result in distribution that are too narrow and we do not have enough coverage of the space
 - Different means Discontinuities – Each latent variable should have a different mean. Don't impose any prior centered at mean 0. We can have vast disconnectedness in our latent space so it is meaningful to traverse a latent space and find point that are similar and related
 - Normal Prior
 - By setting the mean to zero and the standard deviation to 1. We can ensure that over latent variable have overlap and our distribution are not too narrow which will make our regularization space smoother and more complete.
 - center mean regularize variances
 - Regularization with normal prior help enforce information gradient in the latent space
 - Regularization can adversely affect the quality of the reconstruction



Shows non continuity and completeness

VAE Computation Graph

Reparametrizing the Sample Layer

- By introduce the mean and variance and imposing this probabilistic we introduce randomness
 - We must modify the back propagation we can't back propagate gradients due to randomness
- To solve this problem we reparameterize the sampling layer to divert the stochasticity, away from the μ and σ terms then ultimately train the network end to end
- From the picture to the right
 - epsilon is being drawn from a normal distribution
 - We can learn a fixed vector of means a fixed vector of variances and scale those variances by this random constant such that we can still enforce learning over a probability distribution, but diverting the stochastic away from those mean and sigmas that we actually want to learn during training

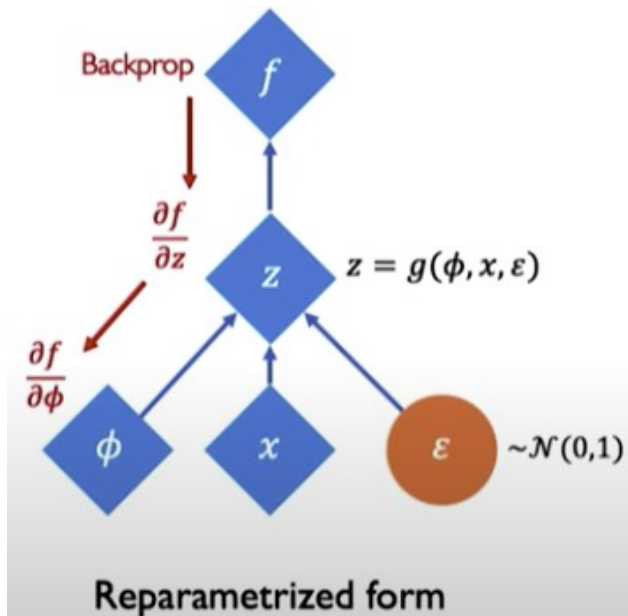
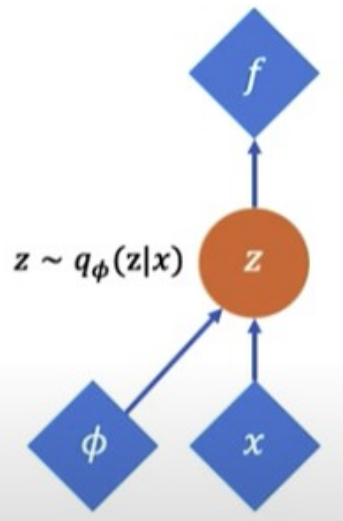
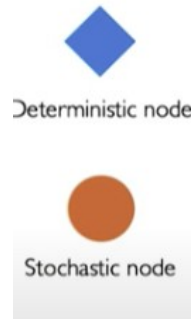
Consider the sampled latent vector \mathbf{z} as a sum of

- a fixed μ vector,
- and fixed σ vector, scaled by random constants drawn from the prior distribution

$$\Rightarrow \mathbf{z} = \mu + \sigma \odot \epsilon$$

where $\epsilon \sim \mathcal{N}(0,1)$

Reparameterizing the sampling layer



diverts the probabilistic operation completely away from the mean and sigma of the latent variables such that we can have a continuous flow of gradient z and train the networks end to end

Variational Autoencoders (VAE)

- For traditional autoencoders we will get the same output for the input
 - deterministic once the network is fully trained
- For VAE – learn a better and smoother representation of the data and generate new images without AE since it was deterministic
 - VAE are stochastic (random)
- Variational autoencoders are a probabilistic twist on autoencoders
 - Sample from the mean and standard deviation to compute latent sample – break down the latent space z
 - Goal of the encoder is to output a vector for both the mean and standard deviation that corresponds to the latent variable z
 - The randomness allows us to generate new data and build up a more meaningful latent space
 - By creating a probability distribution for each latent variable – we will be able to sample from those distributions to create new data
- Over the course of training the encoder is trying to infer probability distribution of the latent space with respect to the input data and the decoder is trying to infer the probability distribution over the input space given that same latent distribution space
 - Get two sets of weights (encoder and decoder)
 - Loss is the reconstruction loss + regularization term (impose some notion of structure in the probabilistic space)
 - trying to optimize the loss with respect to the weights of our network
 - reconstruction loss is mean square error of the pixels

VAEs : Latent perturbation

- Slowly increase or decrease a single latent variable .
 - Keep all other variables fixed.
- With this notion of probability and distributions over a latent variable we can sample from our latent space and tune variable keeping everything else fixed and generate data sample that are perturbed with a single feature or latent variable
- Different dimensions of z encodes different interpretable latent features that we are trying to learn over the course of training can effectively encode and pickup on different latent features that may be important in our data base
- Try to maximize the data we are picking up through the latent variable such that one latent variable is picking up on on some feature and another is picking up on a disentangled (separate) or uncorrelated feature
- Ideally we want latent variables that are uncorrelated with each other
- Enforce diagonal prior on the latent variables to encourage independence

Latent space disentanglement with β -VAEs

Standard VAE loss:

$$\mathcal{L}(\theta, \phi; \mathbf{x}, \mathbf{z}, \beta) = \underbrace{\mathbb{E}_{q_{\phi}(\mathbf{z}|\mathbf{x})}[\log p_{\theta}(\mathbf{x}|\mathbf{z})]}_{\text{Reconstruction term}} - \underbrace{D_{KL}(q_{\phi}(\mathbf{z}|\mathbf{x}) \parallel p(\mathbf{z}))}_{\text{Regularization term}}$$

This is the standard loss function above

β -VAE loss:

$$\mathcal{L}(\theta, \phi; \mathbf{x}, \mathbf{z}, \beta) = \underbrace{\mathbb{E}_{q_{\phi}(\mathbf{z}|\mathbf{x})}[\log p_{\theta}(\mathbf{x}|\mathbf{z})]}_{\text{Reconstruction term}} - \underbrace{\boxed{\beta} D_{KL}(q_{\phi}(\mathbf{z}|\mathbf{x}) \parallel p(\mathbf{z}))}_{\text{Regularization term}}$$

Beta – A hyperparameter control the strength of how strictly we are regularizing.

$\beta > 1$: constraint latent bottleneck, encourage efficient latent encoding \rightarrow disentanglement

Why latent variable models ?

Debiasing

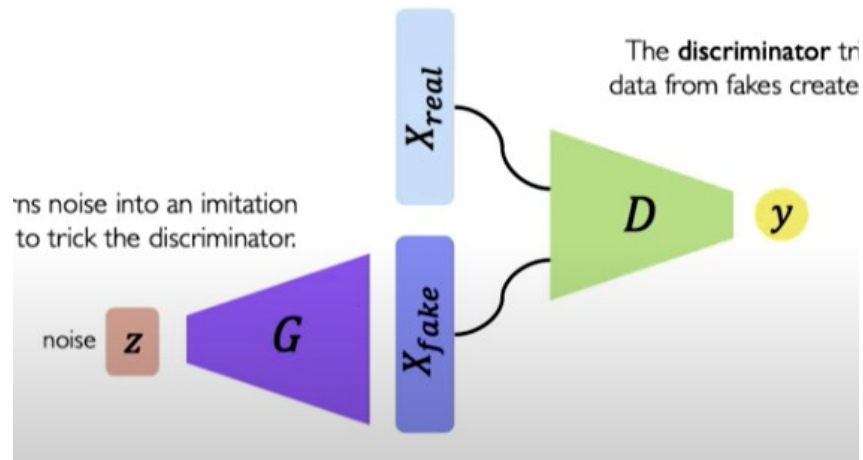
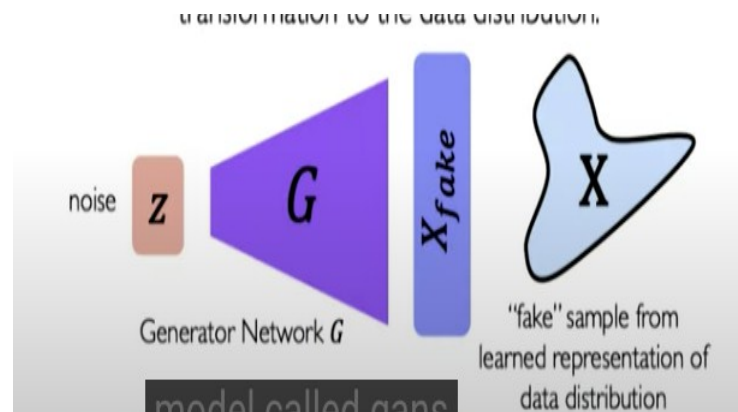
- Capable of uncovering underlying latent variables in a dataset
 - How can we use latent distributions to create a fair and representative
 - Since we are training these networks in a completely unsupervised fashion we can pick up automatically on the important and underlying latent variable of the dataset such that we can build estimates of distributions of the data
 - Use : Adjust and refine our dataset during training
 -

VAE Summary

- Compress Representation of world to something we can use to learn.
- Reconstruction allows for unsupervised learning (no labels)
- Reparameterization – trick to train end to end
- Interpret hidden latent variables using perturbation
- Generating new examples
-
- Key Problem is density estimation – trying to estimate the probability of these latent variables

Generative Adversarial Networks (GAN)

- What if we just want to sample
 - Idea: Do not explicitly model density and instead just sample to generate new instances
 - Problem : Want to sample from complex distribution – can't do this directly
 - Solution : Sample from something simple (noise) learn a transformation (using neural networks) to the target data distribution
- Use the information to sample realistic new instances of data that match our input distribution
 - Problem : Data is very complex and hard to sample new data directly
- GANs are a way to make a generative model by having two neural networks compete with each other
 - The generator turns noise into an imitation of the data to try to trick the discriminator
 - The discriminator tries to identify real data fakes created by the generator
 - Setup a competition between the two neural networks



Intuition Behind GANS

- Generator starts from noise to try to create an imitation of the data
- The discriminator will see the generated samples along with real examples and the task is to calculate the probability if they are real
 - Overtime build up what is real vs what is fake – training the discriminator
 - decrease the probabilities of the fake ones
- The generator will see some examples of real data try to move the fake samples closer to the real data
- Goal : Very hard for the Discriminator to realize what is fake data.
- Training : Adversarial objective for Discriminator and Generator
 - Global Optimum Generator reproduces the true data distribution

Training GANs' Loss Function

$$\arg \max_D \mathbb{E}_{\mathbf{z}, \mathbf{x}} [\log D(G(\mathbf{z})) + \log (1 - D(\mathbf{x}))]$$

Trying to maximize the change of the Discriminator Identifying fake data and real data as real.

The loss of the fake data over the real data is effectively a cross entropy loss of the true distribution and the distribution created by the Generator

Loss for the Generator (G)

$$\arg \min_G \mathbb{E}_{\mathbf{z}, \mathbf{x}} [\log D(G(\mathbf{z})) + \log (1 - D(\mathbf{x}))]$$

Minimize the objective. The generator cannot access the true data distribution so it minimize the Discriminator($G(\mathbf{z})$) instead of the real data

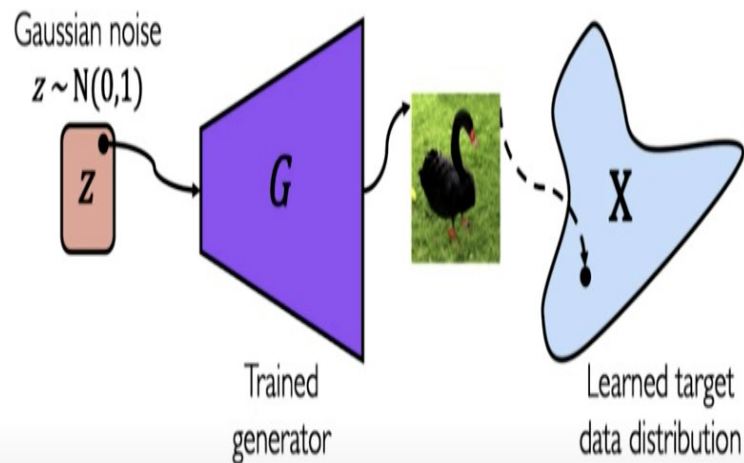
Overall Loss Function :

$$\arg \min_G \max_D \mathbb{E}_{\mathbf{z}, \mathbf{x}} [\log D(G(\mathbf{z})) + \log (1 - D(\mathbf{x}))]$$

Generating new data with GAN

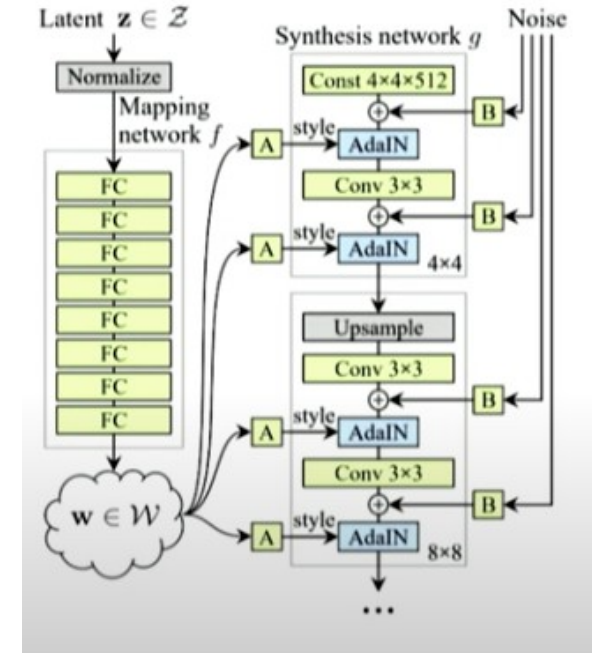
- After training use generator to create new data that never been seen before and sample from
- GANs are distribution transformers
 - One point in the noise distribution will lead to one point in the target data distribution. An independent point will create a new instance in the target distribution
 - We can interpolate and transverse in the noise space to interpolate and transverses in the target space. We can create new data
 -

GANs are distribution transformers



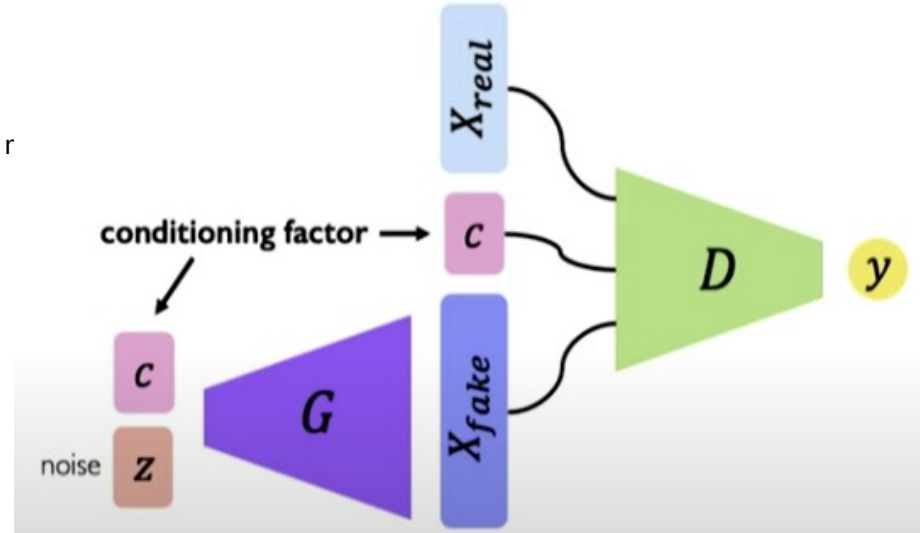
GANs Recent Advances

- Progressive Growing of Gans
 - Effectively add layers to generator and discriminator as function of training.
 - Iterately build up more and more detail image generations
 - Start with a simple model and add more and more network layer to increase resolution
- Style Transfer – Enabled by architecture improvements
 - Build up a progressive growing GAN that can transfer styles
 - Result is remarkable is in the style of the source images



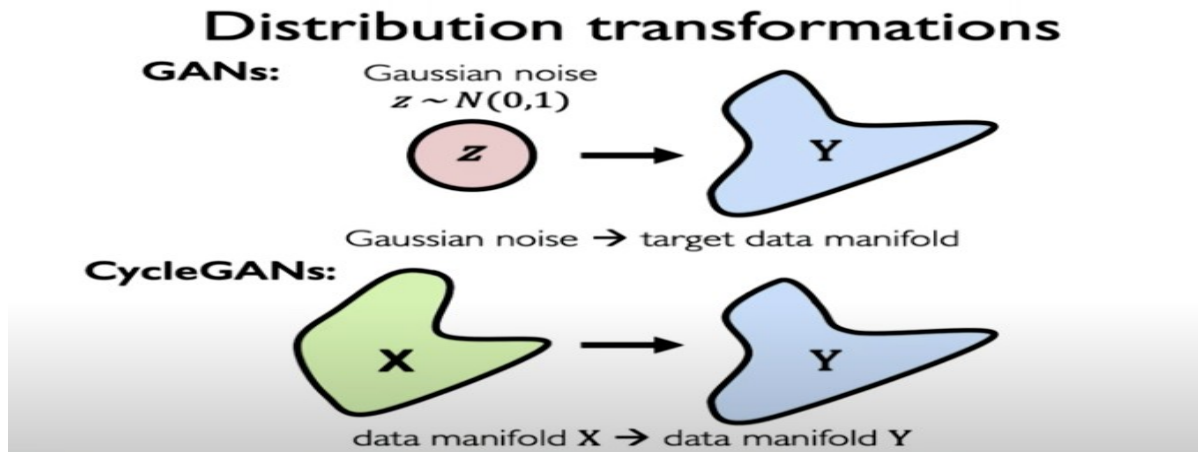
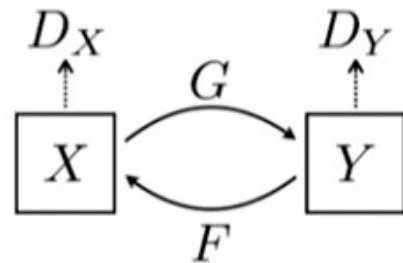
Conditional GANs

- What if we want to control the nature of the output, by conditioning on a label
- Impose a Conditioning Factor (c) to be able to compose generation in a controlled manner
- Application is example paired translation
 - Generating an image of that scene that matches the labels
 - Going from day to night
 - BW to Color
 - Edge to Photo



CycleGAN: domain transformation

- CycleGAN learn transformation across domains with unpaired data
- Now we have two generators and discriminators where they are operating in their own data distributions and learning a functional mapping to translate between the two.
- Ex. Translate from images from horses domain to zebras domain
- Can also extend to Audio and Spectrogram



Summary

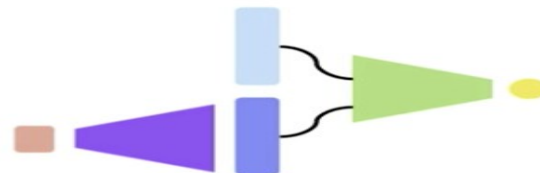
Autoencoders and Variational Autoencoders (VAEs)

Learn lower-dimensional **latent space** and **sample** to generate input reconstructions



Generative Adversarial Networks (GANs)

Competing **generator** and **discriminator** networks

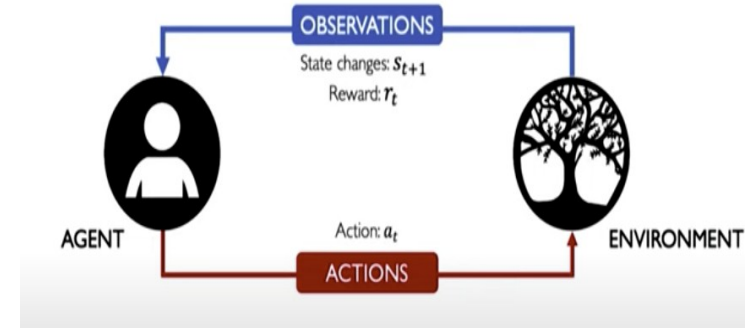


Reinforcement Learning

- Up to now we have been working with a fixed data that we created.
- Now our algorithm will be placed in a dynamic environment so it learns to accomplish its task
- Has implications for robotics and having the computer challenge you to a game (Strategy)
- Classes of Learning Problems
 - Supervised Learning
 - Data (x, y) where x is the data and y is the label
 - Goal – Learn function to map x to y
 - Apple Example : this is an apple
 - Unsupervised Learning
 - Data : x is data, no labels
 - Goal : Learn underlying structure
 - Apple: This thing is like the other thing
 - Reinforcement Learning
 - Data State Action Pairs
 - states – Observations that an agent or players see
 - actions are the behavior taken in the states
 - Goal: Maximize future rewards over many time steps
 - Apple: Eat this thing because it will keep you alive

Reinforcement Learning Key Concepts

- Agent – Anything that will take actions (drone, super Mario)
- Environment – The world which the agents exists and takes actions in
 - Agents can send command to the environment in form of actions
 - Action space A: The set of possible actions an agent can make in the environment
- Actions – Agents choose among a discrete set of actions or continuous set of actions (created by functions)
- When Agent takes actions the environment will send back changes
- Reward – Feedback that measure the success or failure of the agent's action
 - total reward – See picture –
 - discount total rewards – Dampen the rewards effect on the agents choice of action
 - Enforces Short Term or greedy learning
 - Ex. \$5 to take the course or \$5 to take the course and get it in five years.
 -



Total Reward (Return) $\rightarrow R_t = \sum_{i=t}^{\infty} r_i$

Discounted Total Reward (Return) $\rightarrow R_t = \sum_{i=t}^{\infty} \gamma^i r_i$

Defining the Q Function

How to take actions give a Q Function

- Total Reward R_t is the idscounted sum of all rewards obtained from t
$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$$
- The Q function captures expected total future reward an agent in state s, can receive by executing a certain action a
$$Q(s_t, a_t) = \mathbb{E}[R_t | s_t, a_t]$$
- If we are given the Q function
 - How can we choose the best Q Function – ultimately the agent needs a policy $\pi(s)$, to infer the best action to take at its state s
 - Strategy : The policy should choose an action that maximize future rewards : Basic feed each state, action into the Q function and get the max reward

$$\pi^*(s) = \underset{a}{\operatorname{argmax}} Q(s, a)$$

Deep Reinforcement Learning Algorithms

- Two forms
 - value learning – Learn the Q Function and use it
 - How do we get the Q Function
 - policy learning – Directly learn the policy that govern the agent and sample actions from this policy
 - Use the neural network to find or optimize your policy learning and then sample actions from the policy learning



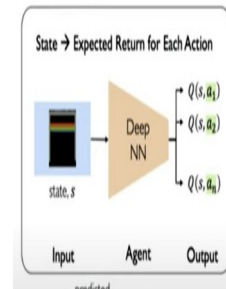
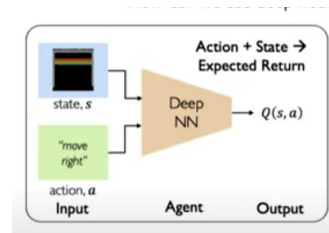
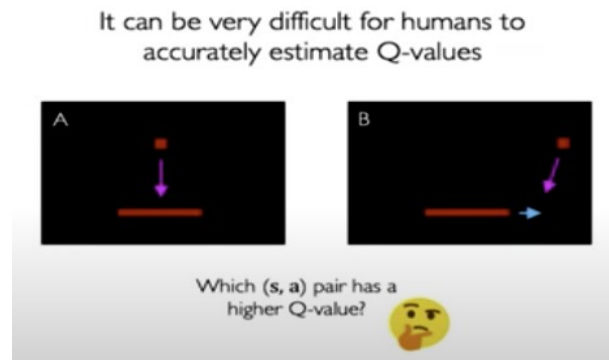
Value Learning

Digging Deeper into the Q Function

Deep Q Networks (DQN)

- The first picture will actually attack the middle and remove one brick at once and slowly solve the problem
- The second will attack the edges and it will be solved faster once it gets above the brick it can knock out multiple
- How can we use deep neural networks to model(learn) Q-functions
 - First approach
 - Input Image input that defines state (convolution input) and action and output the Q state
 - Second approach
 - Input the state and learn to output the Q Value for all different Q Actions. More efficient then the first approach. We only need to run our network once to find the best action
 - What happens if we take all the best actions
 - The target returned would be maximized
 - MSE of the target - the expect value and use the mean squared area

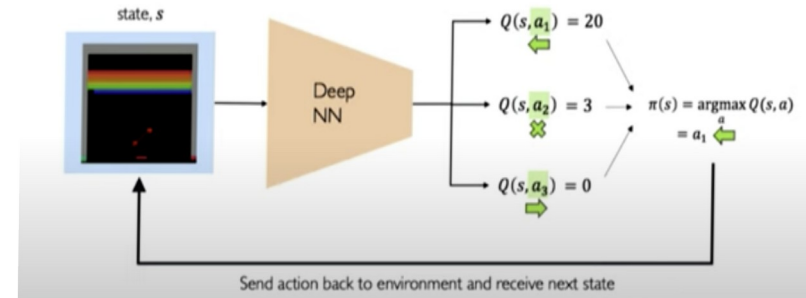
$$\mathcal{L} = \mathbb{E} \left[\left\| \overbrace{\left(r + \gamma \max_{a'} Q(s', a') \right)}^{\text{target}} - \overbrace{Q(s, a)}^{\text{predicted}} \right\|^2 \right] \quad \text{Q-Loss}$$



Deep Q Network

Downsides of Q-learning

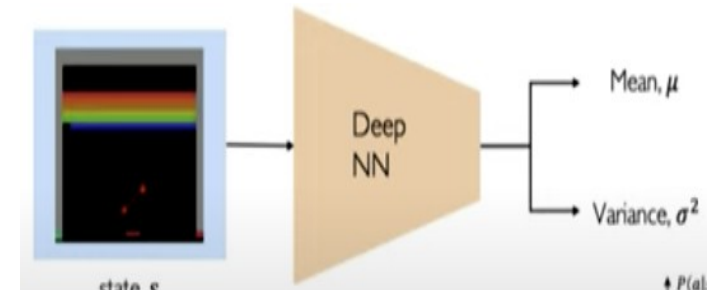
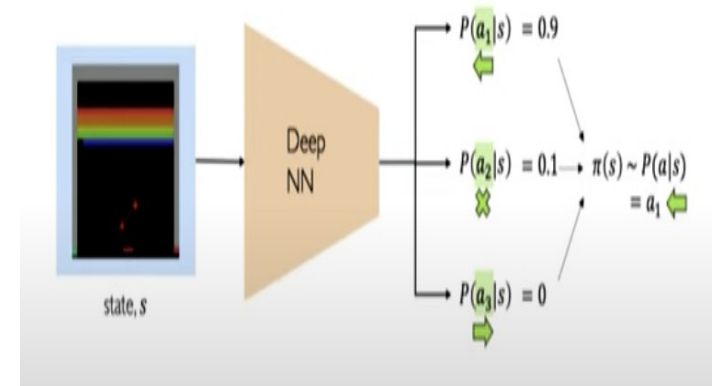
- Use NN to learn Q functions and then use to infer the optimal policy – $\pi(s)$,
- Tested these on many games and found out that 50% of the game played by the computer were able to surpass the human
- Downsides
 - Complexity
 - Can model scenarios where the action space is discrete and small
 - Cannot handle continuous action spaces
 - Flexibility
 - Policy is deterministically computed from the Q function by maximizing the reward – Cannot learn stochastic policy (Environment Can Change)
- The downsides are fixed by policy gradient methods



Policy Gradient (PG): Key Idea

- policy gradient : directly optimize the policy – $\pi(s)$,
 - policy distribution directly governing how the agent should operate and act in this state
 - The output don't give an expected reward, but answer the question if it is a good action
 - To compute the action to be taken sample from the policy function and based on the environment we could get the action that is taken less
 - Since this is a distribution all probabilities must be equal to one
- Advantages to Q Learning
 - Can have a continuous set of action which could include using speed and not just left, right or stay
- Policy Gradient – Enables modeling of continuous action space
 - Ever time we sample, we could see a probability distribution

Policy Gradient: Directly optimize the policy $\pi(s)$



Training Policy Gradients : Case Study

- Cast – Self Driving Cars
 - Agent : Vehicle
 - State (Observations): Camera, lidar
 - Action: Steering Wheel Angle
 - Reward: Distance Travelled\
- Trading Algorithm – The agent should perform better and better
 - Initialize the Agent
 - Run policy until termination
 - Record all states actions and rewards – memory of what happen until it led up to the crash
 - Record all states actions and rewards –
 - When it fails
 - Look at the states/rewards that cam close to the crash – decrease the probabilities of those states, and for action close way before the crash then increase them
- All we gave it was a reward signal and it can be between the lanes
-

Training Policy Gradients : Case Study

- Loss Function
 - From a give state and time the action a will be executed
 - Multiply by discount reward

$$\text{loss} = -\overset{\text{log-likelihood of action}}{\log P(a_t|s_t)} \underset{\text{reward}}{R_t}$$

Gradient descent update:

$$w' = w - \nabla \text{loss}$$

$$w' = w + \nabla \log P(a_t|s_t) R_t$$

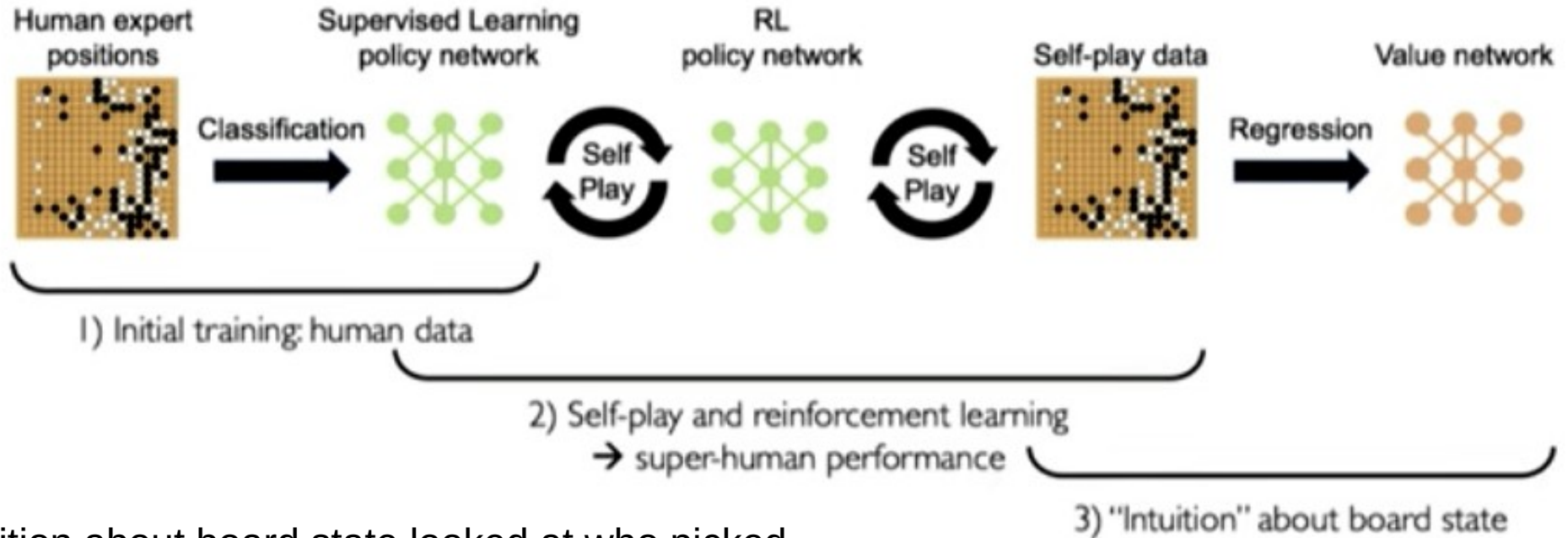
Reinforcement Learning in Real Life

- In real life there are more actions/observations
- Where does the training algorithm break down in real life
 - Run a policy until termination
 - In real life you cannot train your car to drive the streets letting it learn on your own.
- Data driven Simulation for Autonomous Vehicles
 - VISTA : Photorealistic and high fidelity simulator for training and testing self driving cars

Deploying End to End RL For Autonomous Vehicles

- Policy Gradient RL Agent trained entirely within VISTA simulator
- End to End agent directly deployed into the real world
- First full scale autonomous vehicle trained using RL entirely in simulation and deployed in real life

Alpha Go Beats Top Human Play at GO (2016)



Intuition about board state looked at who picked the board

Limitations and New Frontiers

- If you abstract everything away from a neural network it is a function approximator
 - All they are doing is learning a data mapping from data to decision
- Power of neural nets
 - Universal Approximation Theorem – A feedforward network with a single layer is sufficient to approximate an arbitrary precision, any continuous function
 - You have a problem and it can be reduced to a set of input/output that can be related by continuous function
 - Caveats
 - The number of hidden units may be infeasibly large
 - The resulting model may not generalize – How do you find the weights – They may exist
- Limitations
 - Understanding Deep Neural Networks Requires Rethinking Generalization – Experiment
 - For each image they flipped a K-sided Die where K is the number of classes they wanted the label
 - The authors flipped the die for each image and instead taking the true class label – They assigned the label based on the die roll
 - Result : Randomized their labels entirely
 - What happens if we train a neural network on this data set and tested it on a test set where the true labels were preserved
 - The accuracy fell for production, but model was able to get nearly 100% on the training set
 - Means : Modern deep networks can perfectly fit to random data even if it has random labels

Limitations and Frontier

- The neural network is learning the maximum likelihood assessment
 - Where it has observation then we should find an answer
 - Where it has no observations (out of distribution region) – We have no guarantee how the data would look in those regions
 - How do we know when our network does not know
 - How can we establish guarantees on the generalization bounds of our network
 - How can we use this information to inform the training/learning/deployment
- Excellent function approximators only when they have training data
- The quality of the data is just as important the neural network architecture
- Neural Network failure modes Part 1
 - Example – Colorize a black and white go using a neural network and get pink right under his mouth
 - Reasons : In the pictures the tongues are hanging out.
 - Lesson : Deep Learning is powerful at building representations they have during training
- Neural Networks failure modes Part 2
 - Example – What happens when a neural network encounters new data they did not encounter before
 - Found out that some years ago the barrier was not in the data that they used.
 - Need to understand uncertainty in deep learning – When can they not be trusted.

Limitations and Frontier

- How do we build neural nets with limited datasets (Sparse/Noisy/limited) that may contain imbalances
- Uncertainties –
 - Classification of a cat or dog images and output a probability if it is a cat or dog
 - pick a probability of 2 class (cat or dog)
 - The probability of both must be 1
 - What if we have both a cat and dog in the same picture
 - Aleatoric Uncertainty
 - We need uncertainty to assess the noise to the data
 - What if we gave it a picture of a horse
 - With only two class one would have to be low and the other would high since $P(\text{cat}) + p(\text{dog}) = 1$
 - epistemic uncertainty
 - Uncertainty metrics to assess the networks confidence in it predictions
 - Here we are try to capture the model confidence on an out of domain example

