Udacity Deep Learning

Open Q's:

Closed Q's:
- Why do we get new tensors initialized to zeros at the start of each epoch for a LSTM via init_hidden
  - -> Because the memory of the LSTM is RESET after each batch, since then weights have been updated (We reset the hidden state to 0's again but with the same shape)
- Why do we need to change the shape of x after the LSTM cell?
  - The LSTM cell gives us output as (seq_len, batch, num_directions * hidden_size), where hidden_size is the same as hidden_dim or the number of features output has and the num_directions is 1 for unidirectional and 2 for bidirectional
  - We then reshape the vector to get a row for each sequence from each batch with the columns being their respective features
    - -> x.view(-1, hidden_size) if unidirectional
    - ->x.view(-1, hidden_size*2) if bidirectional
- Why is it a problem if we score only according to similarity; Why do we need Queries & Keys in Transformer?
  - By using queries & keys, similar to retrieval systems, we can save some of the computations
  - https://stats.stackexchange.com/questions/421935/what-exactly-are-keys-queries-and-values-in-attention-mechanisms

Chapter 6.6 - Final Project

- Notebook, Train File, Predict File, Index File
- Graduated w/ Passport & NM as name

Chapter 6.5 - Updating a model

- Problem w/ high leel approach: Model does not get created until it needs to be (When you deploy it) -> Low leel approach to modify it
- JSON Object can be thought of as a dict object from Python perspective
- Models return data in different formats -> Refer to documentation
- Combined Model: Define how much data SM should send to each model via Weights
  - -> 'A/B' Test to compare two models & decide which one to keep via update_endpoint
- You are charged for notebook instances on SageMaker; So better make them on your PC and copy over
- Documentation:
  - AWS Doc. /
  - Amazon SageMaker Python SDK / https://sagemaker.readthedocs.io/en/latest/
  - Actual Github Repo -> aws/sagemaker-python-sdk

- **Notebook Instances** provide a convenient place to process and explore data in addition to making it very easy to interact with the rest of SageMaker's features
- **Training Jobs** allow us to create *model artifacts* by fitting various machine learning models to data.
- **Hyperparameter Tuning** allow us to create multiple training jobs each with different hyperparameters in order to find the hyperparameters that work best for a given problem.
- **Models** are essentially a combination of *model artifacts* formed during a training job and an associated docker container (code) that is used to perform inference.
- **Endpoint Configurations** act as blueprints for endpoints. They describe what sort of resources should be used when an endpoint is constructed along with which models should be used and, if multiple models are to be used, how the incoming data should be split up among the various models.
- **Endpoints** are the actual HTTP URLs that are created by SageMaker and which have properties specified by their associated endpoint configurations. **Have you shut down your endpoints?**
- **Batch Transform** is the method by which you can perform inference on a whole bunch of data at once. In contrast, setting up an endpoint allows you to perform inference on small amounts of data by sending it do the endpoint bit by bit.

Chapter 6.4 - Hyperparam tuning

- SM uses Bayesian Optimization
- SM creates a bunch of models & choses the best one
  - -> We tell it how many & what metric
  - Build a hyperparameter tuning object & set params; We can set ranges for the hyperparams
  - Run it -> Attach the best version to a new estimator object (To avoid having to create a new estimator & train it)
- CloudWatch: Lookup logs from specific jobs
  - -> See Deployment Errors; Print custom output

Estimator Object:
- Any Object that learns from Data

LowLevel:
- Define Static Hyperparams in param dict -> Which we will not vary
- Set up training config dict w/ varying hyperparams, resource limits, strategy
- Unique tuning job name w/ <32 chars
- Create tuning job & watch via wait or Tuning job log
- Get Model Artifacts from BestTuningJob

Chapter 6.3 - Deploying a model

- Concept Drift: Underlying data used to train a model might change in the future degrading the model's quality

Deploying a model is just about creating an endpoint – an URL where we can send data to

- High Level Deploying:
  - Call Deploy method w/ no of VMs; Type of VM & create predictor object
  - -> Send serialized data to the predictor object
    - Serialized means e.g. for csv it must be commaa separated
  - Get Predictions back & unserialize (turn into np array)
  - SHUT DOWN Deployed Model, else it will continue to wait for data & u be charged
- Low Level Deploying:
  - Unique endpoint config name
  - Specify Endpoint config info – Name & 'Production Variants'
    - PVs are what model(s) we want to send data to
  - Create an endpoint based on the config w/ unique name
  - Serialize data ourselves (we do not have a predictor object here)
  - Call our endpoint w/ its name; content type & our serialized data

You can see past endpoint configs in the Menu below Inference -> Reuse if needed; Check the URL of the endpoint where you can send data to; Delete it (-> Shuts it down)

Creating a Web App:
2 obstacles:
- How to format the input?
- How to authenticate the WebApp to access our Model?
  - Lambda & API Gateway

Bag Of Words:
- Get term frequency for documents

    - Compare similarity via taking the dot product of the vectors with a 1 for each word for both docs; But Dot P has flaws
    - -> Compare similarity via Cosine Similarity: Divide Dot Product by the product of their magnitudes (Euclidean Norms) -> Equal to angle theta between them
- -> Great for sentiment analysis

Web App:
- Preprocess data to look nice
- Extract BoW -> Only create vectors for e.g. top 5000 words to save space
    - Then each vector has a length of 5000 w/ numbers for each each word (which have int ids)
- Upload the data to S3
- .. Create, Test Model ..
- Deploy! -> SM will set up a VM & get it running
- -> Test the deployed model once more
- Shut down the endpoint
- Once this is done -> We have working model & created an endpoint

- Lambda: Function as a Service: Instead of having a server running all the time; We have a specific function which shall be run, whenever a certain event occurs
    - You only pay each time your function is run

- Lambda Function contains:
    - Preprocessing we need to input the data into our model (BoW)
    - Invokes the endpoint w/ serialized data
    - Read result from model, process & return it

-> We need to give it the permissions, the role to access the endpoint -> IAM
-> Create a new Lambda role & give it SageMakerFullAccess
-> Via Compute Create a Lambda function based on that role & write its code
-> Test the Lambda Function via Test Event (API Gateway AWS Proxy) and putting in the body it would receive

Setting up the API to get the lambda function to work with our web app:
- Amazon API Gateway -> Can be used to trigger different things (We use it for our Lambda function)
    - Create a new API
    - Connect it w/ the lambda

Web App continued:
- Set Up Lambda Function (above)
- Set up API (above)
- Copy Paste API URL in the html & run!

CORS Problems:
- https://medium.com/@dtkatz/3-ways-to-fix-the-cors-error-and-how-access-control-allow-origin-works-d97d55946d9

You are charged per execution for Lambda & API; To keep things clean delete them after a project

Chapter 6.2 - SageMaker

- SageMaker is a combination of two tools
  - Managed Jupyter notebook instance – running on a VM
  - API to supervise computationally expensive tasks
- Training Process
  - Task is constructed on VM
  - Resulting model saved to a file ‘Model Artifacts’
- Inference Process
  - Task is constructed & executed on VM
  - Send data to VM along with Model Artifacts -> Result
- SageMaker capabilities:
  1. Explore and process data
     - Retrieve
     - Clean and explore
     - Prepare and transform
  2. Modeling
     - Develop and train the model
     - Validate and evaluate the model
  3. Deployment
     - Deploy to production
     - Monitor, and update model & data
- SageMaker tools:
  - Ground Truth: Label jobs, datasets
  - Notebook: Create Jupyter NBs etc
  - Training: Choose an ML algorithm, define training & choose hyperparams
  - Inference: Compile & configure; Choose endpoints
- Modularity: You can reuse the model you have built earlier
- SageMaker instance to launch a VM
- Utilization Quotas/Limits on almost all services at AWS
  - Each Quota is regio-specific
  - See them via Service Endpoints and Quotas / Service Quotas Console / list-service-quotas
  - Increase Quotas:
    - Amazon Service Quotas
    - AWS Support Center
    - request-service-quotas-increase
- Stop the notebook, as while it’s running it is incurring a cost!
- XGBoost -> Random Tree Model; Very prone to overfitting
- We need to import the SageMaker module to access their functions
  - Session Object -> Manage data
  - Role -> Defines how the data your notebook uses will be saved
  - S3: Simple Storage Service -> S3 bucket to store data
- High-Level: API we use to get SageMaker to perform various ML tasks
- Low-Level: A lot more details we must take care of
- get_image_uri -> Given the region name & algorithm we want to use, creates the appropriate container
- Estimator Object: A Wrapper for our algorithm
  - Specifies some details for how an object will be trained

- Training Job – Train a specific estimator; You need to provide:
  - Location on S3, where training & val data will be stored
  - Location on S3, where resulting model will be stored (Model Artifacts)
  - Location of Docker Container to be used for training
  - Description of compute instance to be used
- Transformer Object to Test
  - Create a Transformer Object;
  - Call it with the location, contet type & split method
  - -> Call the wait function to see the progress
  - -> Once the Inference has happened SageMaker stores the file w/ the predictions on the S3
- Clean up data directory at the end to avoid full disk space
  -

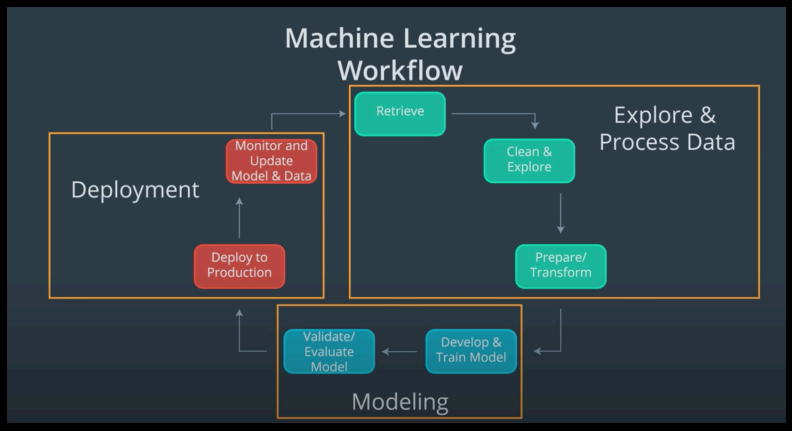
In Depth:

- S3 – Data Storage Buckets -> You can navigate and see what's in them via the Menu -> S3
- URI: Uniform Resource Identifier
- Creating the Model:
  - High Level - Creating a training job:
    - Get the URI to the xgboost (algo) docker container
    - -> Create an estimator object with the algorithm (the container just created); Info about type of VM; Where to store the ouput; Role of VM; additional session info)
    - Set Hyperparams
    - Call fit method w/ link to training & val datasets
  - Low Level:
    - (Specifying all bits of info in one big dict object…)
    - Get the URI
    - Specifiy Role
    - Specify Algorithm via URI
    - Specify output path for model artifacts
    - Properties of VM
    - Set a stopping condition (in case VM gets stuck in a loop)
    - Set Hyperparams specific to algo we chose
    - Tell SageMaker where to pull the data from
      - -> Where training, val data is & their types
    - Create unique name of training job
    - Feed in params & run
    - Logs for jobs method to get updated when finished & see progress

- You can see your past training jobs in the Menu!
  - -> You can acess it; Scroll down; View Logs for potential debugging

- After training has finished: Build the Model in SM
  - A model is a collection of data: The Model Artifacts & Add Info on how to use those
  - Model name must be unique
  - Create container w/ model artifacts & the algorithm container
  - -> Create Model w/ name of model; role & container

You can see your models in the Menu!

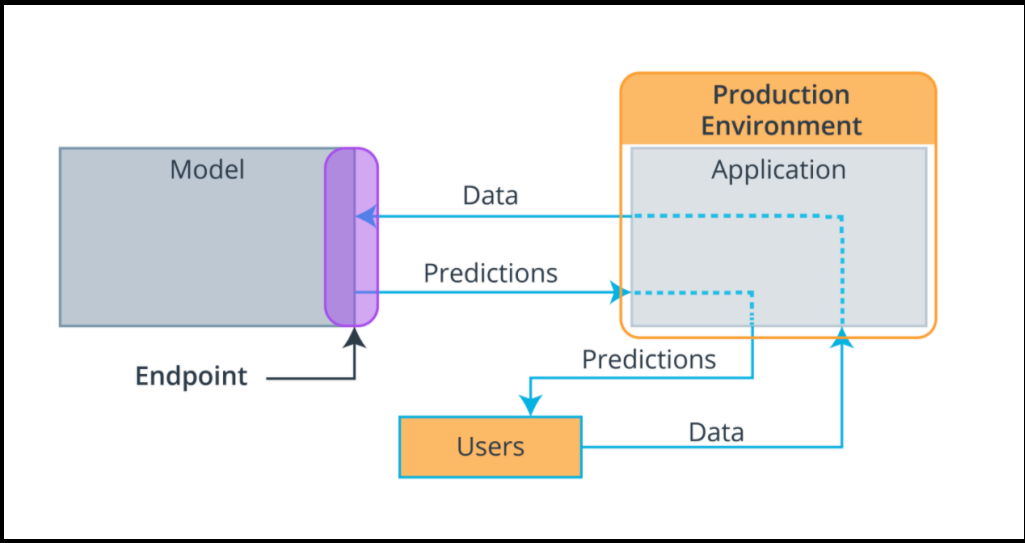


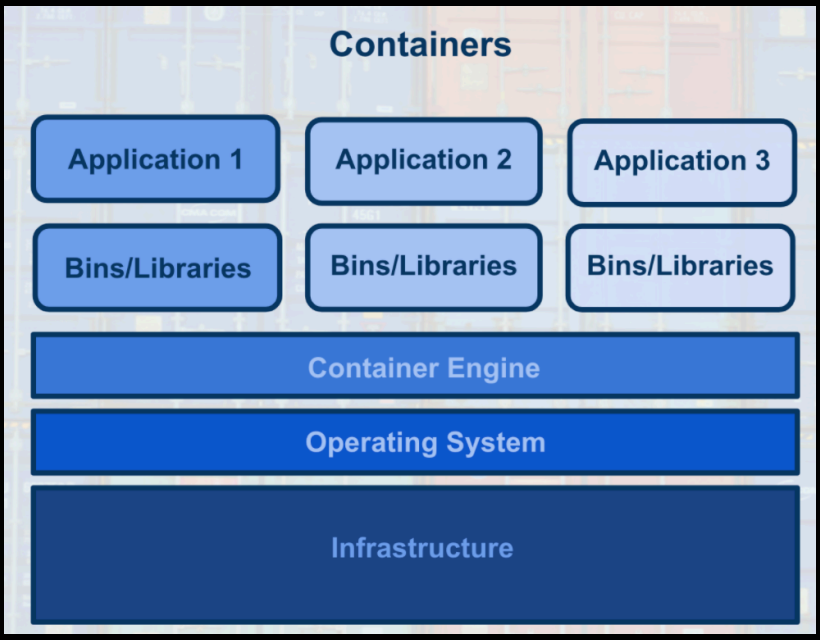
- Test the model:
  - High-Level:
    - Create transformer object to describe no & type of VMs
    - Create batch transform job -> then executed by SM
  - Low-Level:
    - Unique Transform Job Name
    - Describe Transform Job -> Name; Model; How many simultaneous transforms; MaxDataSent to model at given time; BatchStrategy -> MultiRecord (multiple records in each batch of data); Output path; Input; Transform Resource
    - Create the job; Call wait to see progress

Chapter 6.1 - Deployment

- Cloud computing turns an IT product into an IT service
  - Cloud Storage
  - Cloud Applications
  - Databases
  - Virtual Machines
  - SageMaker ..
- Benefits - It's about time & cost:
  - Proportional costs; Fixed cost into Variable
  - Scalability
  - Availability / Reliability
- Risks
  - Security Vulnerabilities
  - Reduced Governance
  - Limited Portability between Providers
  - Multi-regional compliance & legal issues
- Three paths to deployment
  - Python model is recoded in programming language of production environment
    - -> Often Java / C++; Too time-consuming
  - Model is coded in PMML / PFA Languages
  - Model is converted into format that can be used in prod. environment (most common)
    - Methods within Torch/TF etc to convert into a industry standard such as ONNX -> This is then converted to the software native to the production enviroment
    - APIs help ease the work to be done
- Deployment also called 'Operations' & The other two as Development
  - Traditionally handled by diff people, but these days can be done by one as very easy
- The application (interface) communicates with the model via an interface called endpoint
  - Endpoint is like a function call
  - The function called is the model

- The python program where everything is written is the application
- The endpoint API is based on a REST architecture (Representational State Transfer)
  - Uses HTTP requests to enable communication btw application & model via the endpoint
  - -> Followed by HTTP response
- HTTP request composed of four parts:
  - Endpoint in the form of an URL
  - HTTP Method (POST in this case -> used to create new information, which is sent back)
    - GET would only retrieve info; UPDATE; DELETE also have diff purposes
  - HTTP Header – Additional info e.g. format of the data, that's passed to receiving program
  - Message/Data/Body – The user's data to input into the model
- HTTP response three parts:
  - HTTP Status Code (e.g. 200 for all received and good)
  - HTTP Header – Add. info; format of data etc
  - Msg/Data/Body – Returned Data/Prediction
- Application must
  - Transform user data into format easily packed in HTTP request & ready for input to model
  - Transform predictions fro m HTTP response to be displayed/used
  - Often CSV/JSON for both
- Both Model & Application require a computing environment -> Containers for each
  - Standardized software used only for running an application
  - Most popular is Docker
- Container Structure
  - Inf: Cloud providers data center / Local computer
  - OS: OS on your local comp
  - CE: e.g. Docker Software
  - Bins/Libs – required to launch, run & maintain the app layer
- Advantages of cont. architecture:
  - Isolates the application –> Better security





    - Requires only software needed to run the app
    - Easier creation, replication, deletion of app
    - Easy to save & share containers
        - Via container script file -> DockerHub
- Cloud Platforms for ML often provide methods for automatic hyperparam tuning – if not do it via scikit learn
- 4 characteristics of Deployment platforms should have
    - Versioning: Should be able to indicate deployed model's version on the platform
    - Monitoring: Easy monitoring
    - Updating & Routing: Upload new vers. of model ;;; Ability route user data to diff versions of the model to compare
    - Predictions:
        - On-demand: -> Real time
            - Low latency response time
            - Able to handle variety in request volume
                - User request's size may be limited by Cloud Platform (e.g. 5MG by sagemaker)
        - Batch:
            - High volume of requests in periods
            - Common in businesses – e.g. a weekly customer satisfaction prediction
            - Responses & requests often stored in files on the Cloud's providers storage cloud

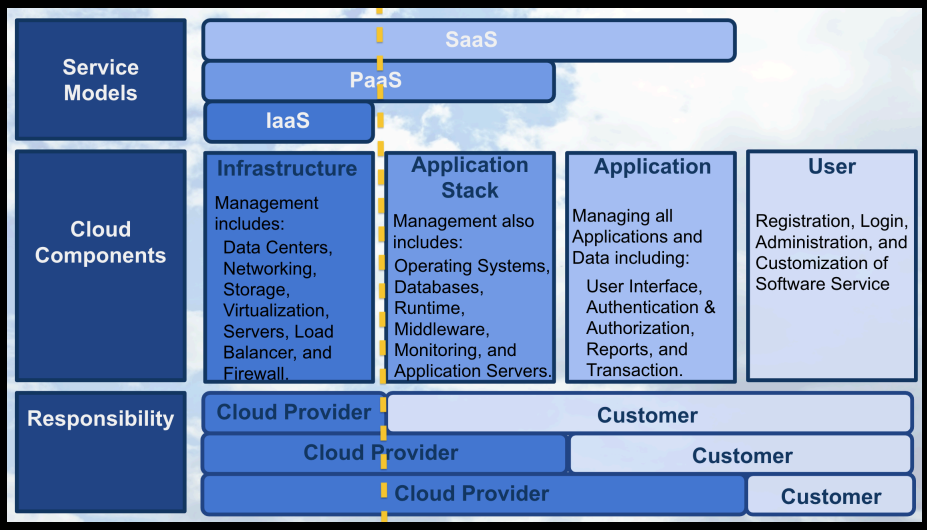
- Comparing Providers:
- https://medium.com/@robaboukhalil/a-tale-of-two-clouds-amazon-vs-google-4f2520516a38
    - AWS - SageMaker:
        - Flexibility across many ML Softwares
            - Built-in algorithms for classification
            - Custom algorithms can be done w/ Torch, TF, Apache, MXNet, Chainer
            - Own algorithms you code yourself
        - Compatible w/ Jupyter NB
        - Flexibility in Modelling & Deployment
            - Automatic Hyperparam tuning
            - Monitoring & Routing possibilities
            - On-demand/Batch predictions
    - Google - GCP ML Engine:
        - Prediction Costs: w/ SageMaker you must leave ressources running to provide predictions; while GCP allows you to only turn on the model when a request is made
        - Exploring Data: Datalab instead Jupyter NB; Google separates Data exploration, processing and transformation into diff. services
        - Less flexibility w/ ML Software: Mostly TF & Scikit learn
        - Equally flexible in Modelling & Deployment
    - Microsoft - Azure AI
        - All frameworks possible - torch etc
    - Paperspace
        - Simple GPU backed VMs w/ all frameworks
    - Cloud Foundry
        - Open Source Cloud Application

-> It depends a lot on your needs though and for how much of the ML process you plan to use the cloud

-> Google focuses only on the Deployment part (hence you can still use Jupyter B etc on your computer as you want), while in AWS you can do all things on the cloud

Three Service Models of Clouds:

- SaaS: WiX, Shopify (You do no need to manage the third column of compoents)
- PaaS: Easily build, host & monitor an application -> Heroku
- IaaS: AWS etc



Four Deployment Models:

- Public Cloud: Can be used by general public; Offered by AWS, Google, Azure – Often they allow for a virtual private cloud tho
- Community Cloud: AWS GovCloud -> Limited to people who have been granted access; Computing resources are isolated
- Private Cloud: Exclusive Use by single organization; Maybe physically located at company
- Hybrid Clouds: Mixed

Five Essential Characteristics a service must have to be considered Cloud Computing:

- On-demand self service: Modifying files on cloud w/o need of an employee
- Broad Network Access: Access cloud files w/ any device connected to internet
- Resource Pooling: Dynamic re-assigning of virtual resources to serve multiple customers
- Rapid Elasticity: Customers can store just one or thousands of images
- Measured Service: Resource usage can be monitored & reported for appropriate billing

Chapter 5.4 - Implementing CycleGAN

- Residual Block: Avoiding Deep Neural Networks from stopping to learn by mapping not from x to y, but from x to x + residual function
  - This residual function is easier to optimize than a direct mapping
  - In general, skip connections to preserve much earlier input help to avoid the learning to stop
- LS instead of CE for our loss function as for very big values the loss is still very high, while for the Sigmoid CE, it vanishes -> Vanishing Gradient
  - Other options are e.g. the Wasserstein GAN
  - LS works despite the fact that there is no sigmoid and only Conv Layers, as the Determinator will learn a representation of real images of its own minimizing the given loss function – as the Loss function is D_out - 1 … it will learn to get a d_out close to 1 in its final conv layer for real images and close to 0 for fake images; The Generator will try to turn this around and get close to 1 as well by using the same loss function

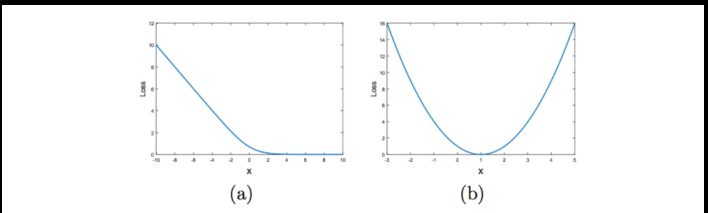
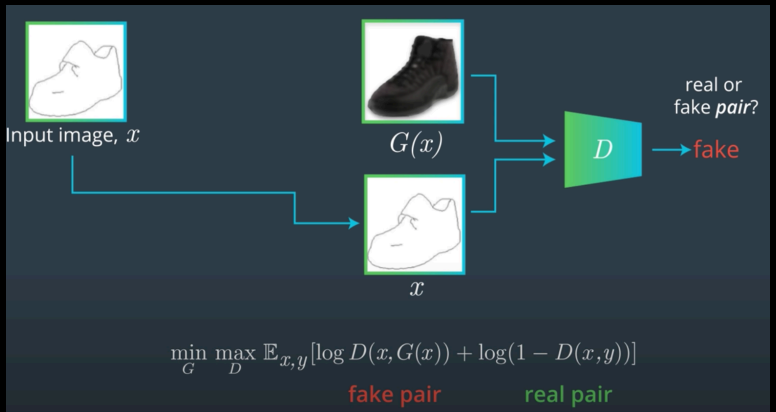


Figure 2: (a): The sigmoid cross entropy loss function. (b): The least squares loss function.

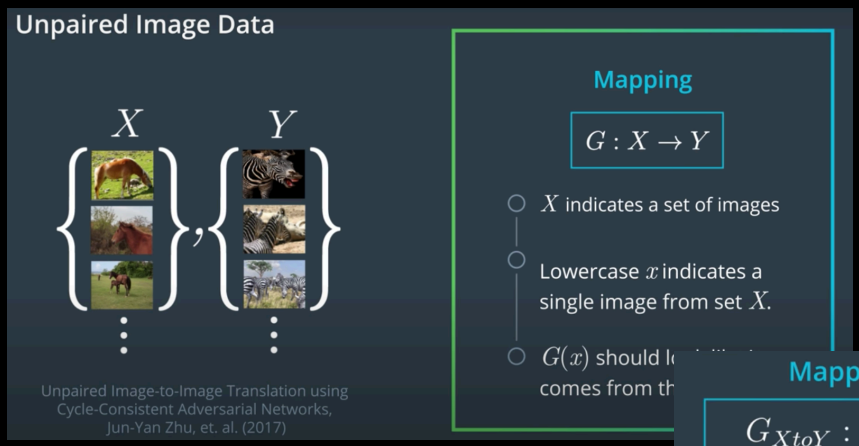
Loss patterns for large `x` values. Image from the LSGAN paper.

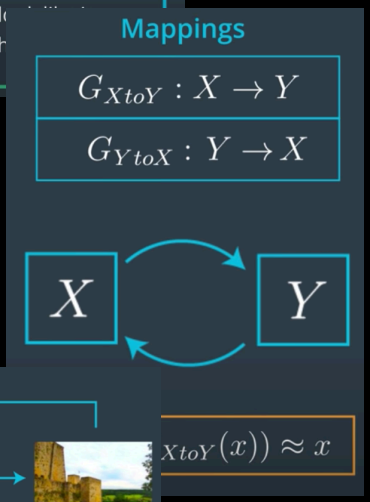
Chapter 5.3 - Pix2Pix, CycleGAN

- Pix2Pix -> Give the GAN more accurate images e.g. a drawn shoe & turn into a real shoe
  - Encode the drawn shoe via multiple conv & Batch norm layers in the generator
    - This will get us a small feature representation of the input image
  - & then decode it again -> Same as normal GAN;
    - Take the feature level representation & via Transpose Convs turn it into an image
    - To ensure the D only accepts

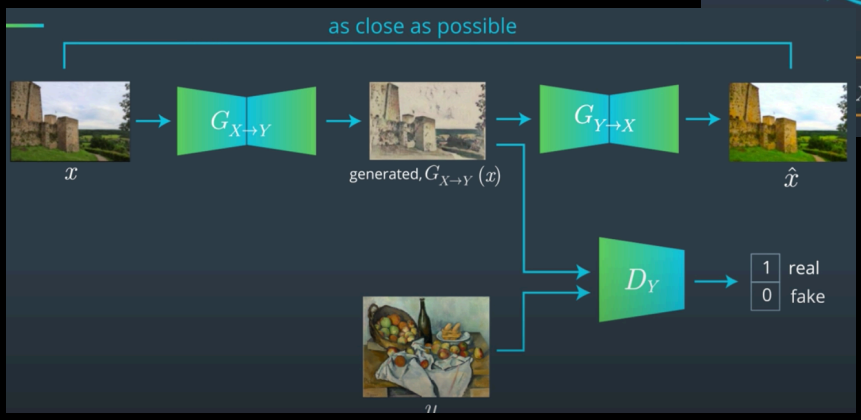


images based on the drawing, we need to connect the drawing as well to the D and give him a pair of two images and then let him determine whether this image was generated based on the drawing or is real
    - Problem: Collecting REAL Pairing data is very difficult and epensive (You need e.g. a drawing & a real image) -> Cycle GANs





- Pix2Pix Use Cases:
    - Aerial to Map (Google Maps)
    - Black/White to Color
    - Edges to Photo
    - Day to Night
    - Labels to X
- Cycle GANs
    - Turning a horse into a Zebra is impossible w/ Pix2Pix, as you cannot get a Zebra to do the same pose in the same environment as a horse
    - Creating a complete Transformation Cycle!
        - -> Similar in Translation: Translating an English Sentence to French & then French back to English should get you the same English Sentence
    - ‘Cycle Consistency’
        - Difference between x and x_hat is the forward cycle consistency loss, also called the reconstruction error
        - Diff between y and y_hat is the backward consistency loss
        - The complete loss is then the two Adversarial losses + (forward cycle loss + backward cycle loss) * lambda
            - Lambda weights how important the consistency is
        - Problems:
            - Only shows one version of possible output even if there are multiple
            - Produces low resolution images
            - It may fail..
        - Promising Advances:
            - StarGAN
            - One-to-many; Many-to-many



Chapter 5.2 - Deep Convolutional GANs ('DCGAN')

- Discriminator: No Maxpools; Downsampling only via stride=2
- Generator: Transposed Convolutions to upsample
- Internal Covariate Shift: When the inputs to each layer are different in their distribution
  - Solved w/ Batch Norm.
    - Calculate Average at each input step (mu)
    - Calculate StD at each input step via sq root of Variance
    - Then normalize via Formula on PICTURE
      - epsilon tiny no to avoid ZeroDivisionError
      - & to slightly increase variance as we want the variance of our total population and since we only have a batch, our variance is biased to be slightly lower than the real one
      - Take the new inputs & to give our network some more knobs to turn we scale it by another weight gamma & bias beta
      - Then pass the normalized values into the activation function

$$\hat{x_i} \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y_i \leftarrow \gamma \hat{x_i} + \beta$$

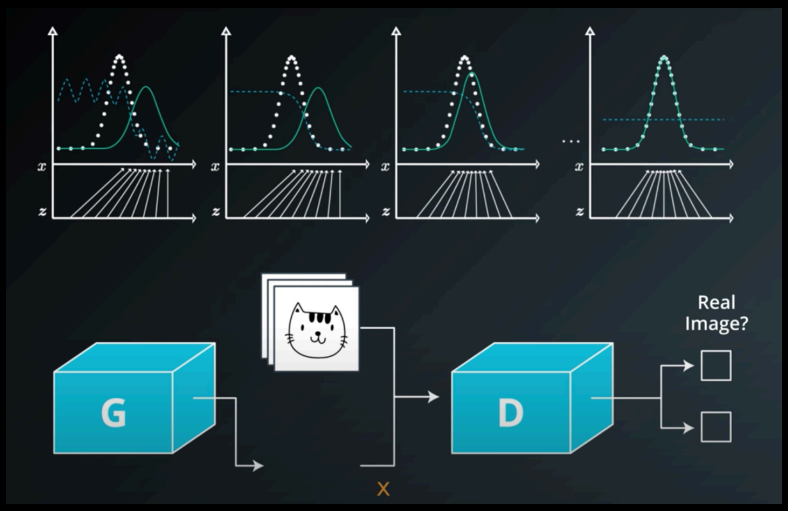
    - BatchNorm1d for Linear Outputs; BatchNorm2d for 2D like filtered images
      - Set bias=false for the linear layer preceeding the batch norm, since we already have the bias term in the batch_norm
      - When you set to eval; the model will use the mena & variance of the whole population not just batches
    - BatchNorm does also work w/ RNNs & ofc CNNs via bn2d — look at the linked papers in the notebook if necessary
    - Batch Norm BENEFITS:
      - Faster Converging
      - Allows higher LRs
      - We can be less careful w/ weight initialization
      - Better for Activation functions
        - By regulating the values going into the AFs, they are less likely to to loose their gradient in deep NNs due to outliers
      - A bit of regularization due to extra noise
      -

Chapter 5.1 - GANs

- Applications - mostly images:
  - StackGAN: Textual description -> Generate image
  - iGAN (Berkeley & Adobe): Drawing -> Real image
  - FB AI: Photo -> Cartoon

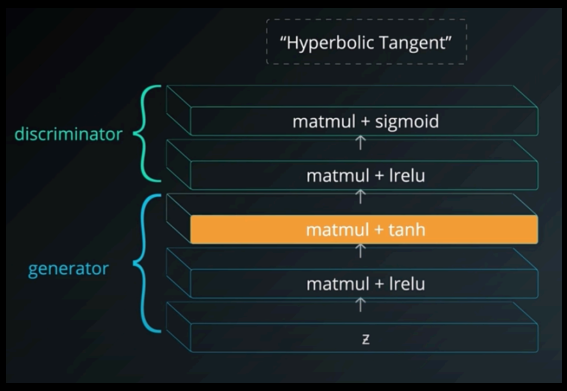
- NVidia: Day Scene Photos -> Night scene photos
- Cycle GAN: Unsupervised Image Translation
- Apple AI: 3D rendered images of eyes -> Real Eyes -> Use them to train for Face Recog.
- Imitation Learn: -> Learn to imitate actions from a human expert
- GANs to predict outcome of high-energy particle physics; What outcome is likely to occur in each situation
- Produce Adverserial examples – Images that look perfectly normal to a human, but are designed to fool a machine learning classifier

- RNNs can also be generative models, generating one word at a time
- GANs can also generate one pixel at a time (Fully visible belief networks / Autoregressive models)
- But GANs can also generate a whole image in parallel
- Generator Network takes random noise as input; Runs it through a differentiable function & reshapes it to have recognizable structure
  - Choice of random input noise determines which image will come out of GAN; Many different input noise values produce many different realistic output images
- Training is very diff. from traditional NNs, as there is no correct output
  - We give the GAN a lot of images and ask it to make images coming from the same probability distribution; But difficult to compute the exact probabilities to make a REAL image
  - -> A second network, called the discriminator learns to guide the Generator; It's a regular neural net classifier, which classifies images as real or not
    - It is shown real images from training data & fake generated images from the GAN half of the time; Outputs the probability that a image is real
      - -> The Generator tries to fool the discriminator and output images which will get a near 1 probability from the discriminator
      - The Generator maps noise images (z) to real images (x); it tries to move uphill on the determinators probability distribution to get higher values for its images -> Eventually the prob. distribution of Generated images equals the white line (prob distribution of real images); D will be forced to output 0.5 for each image as it does not know whether real or not anymore



    - Equilibrium: Neither player can improve their payoff by changing their strategy, assuming the other players strategy stays the same
    - Usually we don't find a global minimum in loss functions as the optimization algorithm gets stuck at some very low point
    - GANs are diff, as there are two players and each player has its own cost function
    - Simple case: Cost of D is negative of Cost of G
    - The Equilibrium will be a saddle point -> The local minima for one cost function & local maxima for the other
    - GANs do not normally find the perfect equilibrium
      - Common failure: G learns to generate one cluster from real image -> D learns to label that cluster as fake -> G deletes that cluster and takes a new one -> Instead it should sample from all real clusters

- Architecture is key for good GANs
  - Simple task we can get away with a fully connected architecture
    - Both generator & discriminator should have at least 1 hidden layer
    - Leaky ReLUs work the best -> Gradient can flow through entire architecture
      - As the generator receives a gradient from the discriminator
    - Output of generator: Hyperbolic tangent
      - -1 to 1
    - Output of discriminator: Sigmoid
  - Two optimization algorithms simultaneously
    - Adam optimizer is good for both
  - BCELoss (Sigmoid Cross Entropy Loss)
    - You want to use the logits! The values before the sigmoid activation function
      - As otherwise there might be rounding errors when sigmoid is cloe to 0 / 1
    - Multiply zero/one labels with a number close to 1 (e.g. 0.9) to avoid extreme predictions and help the determinator generalize better
  - Generator Loss: Same but with flipped labels
    - Do not use simply the negative d_loss, as the gradient of d_loss is zero whenever the grad wins -> The generator would try to maximize Cross Entropy, but we want both to minimize a function of their own



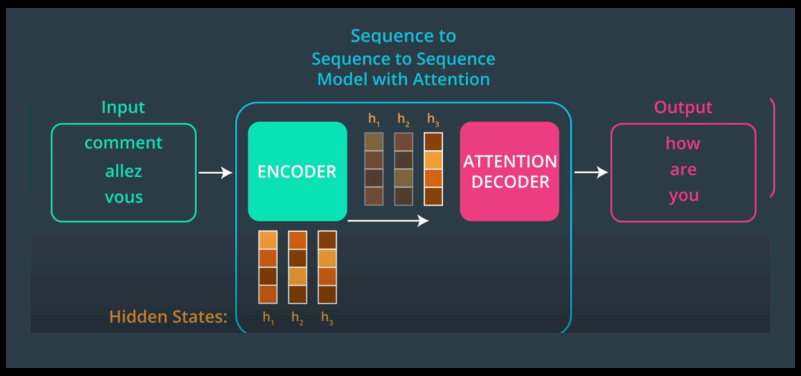

```
d_loss = nn . BCEWithLogitsLoss (logits, labels*0.9)
g_loss = nn . BCEWithLogitsLoss (logits, flipped_labels)
```

- To scale up to large images -> CNNs within
  - For the Generator we want an 'opposite' CNN, as in we start small and then expand the features and the image e.g. via CNN Transpose operation (similar for Autoencoders)
- Use Batch Normalization (DCGAN paper) on each layer except ouput layer of generator & input layer of discriminator
  - https://video.udacity-data.com/topher/2018/November/5bea0c6a_improved-training-techniques/improved-training-techniques.pdf
- BCELoss applies Sigmoid & then Binary cross entropy & is more numerically stable than applying both separately!
- Latent space: How objects are mapped to each other in the hidden dimensions

Chapter 4.8 - Attention

- Focusing on certain things in ML models
  - E.g. for CNNs - only few parts are often necessary to identify an object
  - Neural machine translation: Only focused on words one at a time
- Sequence to Sequence:
  - Input into Encoder -> Transforms input into a context vector -> Decoder -> Output
    - Context Vector has normally length of 256 / 512

- -> Problematic since for long sequences there might be more info to process
- If you increase the length of this context vector (which is the #3 hidden state), then you overfit on short sequences!
- -> Thats the problem Attention solves



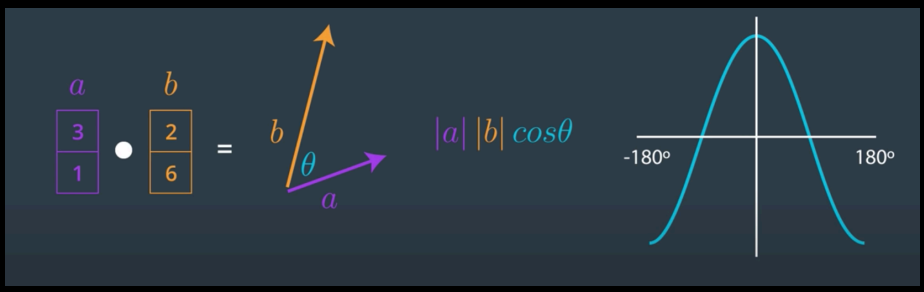
- The attention architecture is able to translate different sentence ordering by learning skipping certain hidden states and put them in later
- Attention Decoder
  - Scoring Function to determine score for each hidden state
  - Feed scores in to softmax -> All positive & sum up to one
  - Multiply each hidden state vector by its score & sum them up to produce an attention context vector
  - We calculate one of those attention context vectors at each time step in the attention decoder
  - Formular for additive attention
    - First line is about calculating a single number for each decoder timestep taking as input the hidden state (h) and the previous state (s) and three weight matrices v, W, U
    - Second line passes it intop Softmax
    - Third line turns it into weighted sum average -> The attention context vector

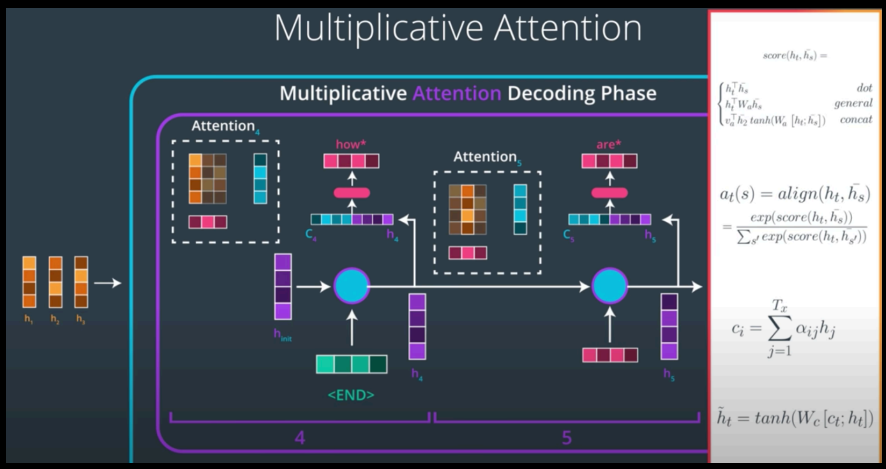
$$e_{ij} = v_a^\top \tanh(W_a s_{i-1} + U_a h_j)$$

$$a_{ij} = \frac{exp(e_{ij})}{\sum_{k=1}^{T_x} exp(e_{ik})}$$

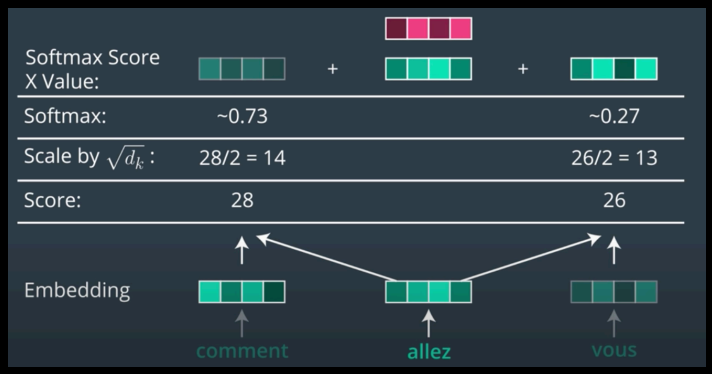
$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$$

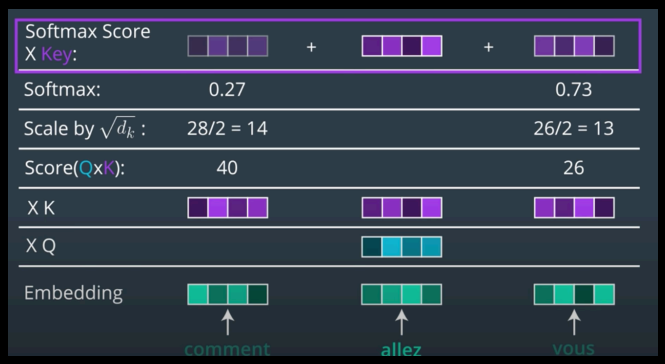
  - Multiplicative attention
    - Stacks both in the encoder & decoder function
      - -> We pass in only the hidden state from the top RNN layer
    - three different scoring function (dot, general, concat)
    - Attention context vector calculation is the same
  - The dot product is the same as multiplying the length of two vectors and the cosine of their angle in between; the cosine is exactly 1 at 0 degrees
  - -> The more similar the vectors (similar angle) the larger the dot product - 'Similarity Measure'
    - The problem with the dot product as scoring function is that the encoder & decoder must have the same embedding space -> We might want a different embedding space for English vs Chinese
    - The 2nd scoring function





introduces a Weight Matrix between the dot product
- Third scoring method is essentially a small neural network
- h(t) = target -> hidden state of decoder
- h(s) = source -> hidden state of encoder
- Attention models put out words based on what they see and then gradually see more / change their focus
- The Transformer -> Paper Attention is all you need -> Removing RNNs & only having attention; All of these attention layers are similar to multiplicative attention
  - Stack of identical encoders & decoders (paper proposes 6 for each)
    - Each of these 6 encoder layers contains two sublayers
      - Multi headed self attention layer
      - Feed Forward layer
    - 6 Decoder layers:
      - Two attention components
        - Encoder Decoder attention (Focuses on the current input)
        - Self-attention (Pays attention to previous decoder ouputs)
      - Feed forward
  - Recap on self-attention: Current embedded word is allez -> We look at the other hidden states and how comparable they are (comment gets a higher softmax function here so is more important as it had a higher score, hence higher 'similarity') -> Then we get our final sum for this word
  - In practice however we do also need to add Queries & Keys, as with only embedding the calculations would only focus on the similarity of words





Chapter 4.6 - Sentiment Prediction RNN

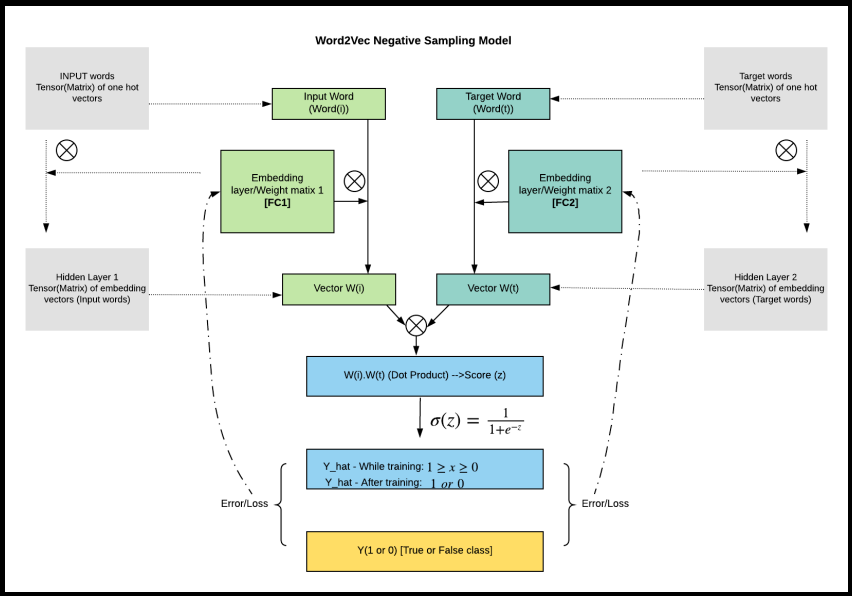
- Embedding layers purposes:
  - Learn semantic representstions of words (the vectors)
  - Reduce dimensionalities (more efficient than one hot encoded vectors)

- For the former we can train it separate from e.g. our real model such as LSTM
- We are only interested in the last sigmoid output of the review (not the output from each word; only after the whole review / sequence)
- Bidirectional LSTM -> We also use the text after the target word (The boys went … and partied)
  - Unidirectional – we only use what's before (The boys went…)
  - Bidirectional can be used for sentiment analysis when you already know what comes after and will perform better
- BCE - Binary Cross Entropy -> Applies Cross Entropy to a single value btw 0 & 1
- For Script generation generally good to have >1MB of script text

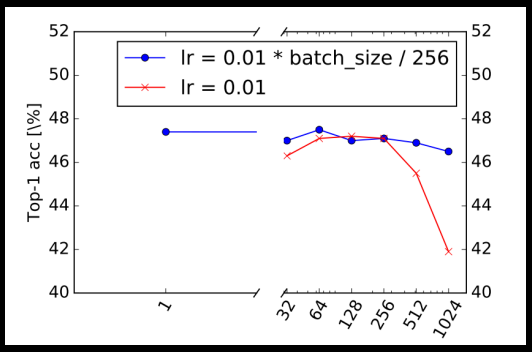
Chapter 4.5 - Embeddings & Word2Vec

- The relationship btw the embeddings for walking & walked should be the same as for swimming & swam
- These embeddings are learned from source text -> Ensure source text contains correct information
- Embedding Layer:
  - Pass in word as an integer, e.g. heart as integer 958
    - Take the value of the 958th row of the embedding weights matrix
    - Those weights in the embedding matrix are learned
    - Embedding dimension (columns of the embedding matrix) is how many values our vector for that word has; normally in the hundreds
      - It is representative of how many features we will be able to detect
- Implementing Word2Vec:
  - CBOW: Giving a sentence and have it predict the missing word
  - Skipgram: Give it the word and have it predict the context (Better)
    - E.g. The output could be two words before and two words after
- Cosine similarity
  - The similarity between two words is the cosin of the angle between them
  - When the angle is 0, the similarity is equal to 1 (The maximum value cosin can take)
  - When the angle (theta) is 90 (the vectors are orthogonal) then the cosin is 0
- TSNE – T distributed Stochastic Neighbor Embeddings
  - Non Linear dimensionality reduction technique to cluster data
  - http://colah.github.io/posts/2014-10-Visualizing-MNIST/
- Traditionally takes very long to train; Methods to reduce
  - -> Negative Sampling
    - Traditionally we update the weights to ALL other words (although we only care about few other words for a given word)
    - In negative sampling we only update the weights to the target words and a few others (sampled at random)
      - Introducing 2nd embedding layer

Chapter 4.4 - Hyperparameters



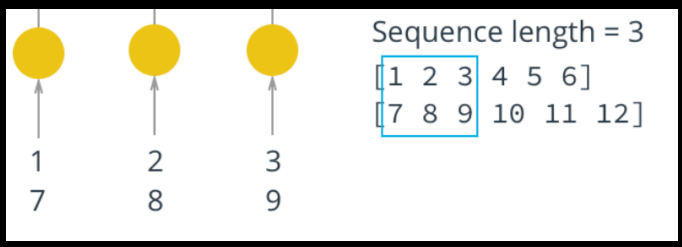
- Minibatch size vs learning rate for imagenet it seems as 128 & 256; about 1/4 of the total classes is a good batch_size
- https://arxiv.org/pdf/1606.02228.pdf
  - Great paper on comparing diff. hyperparameters
- Neurons: Tedency to increase vs decrease (Not more than twice the amount of inputs)
- Hidden Layers: 3 will always outperform 2 according to Karpathy; Afterwards little change (Except for CNNs)
- RNNs: LSTM & GRU always better than trad. RNN
  - https://arxiv.org/abs/1506.02078
  - Try both – very similar in efficacy
  - 2 Layers is good; 3 Layers rarely help (except for speech recognition)
- The embedding size is how many features we give our inputs (e.g. with a hot encoded vector on the alphabet we have ~26); But we could also use word2vec or hot encode words instead

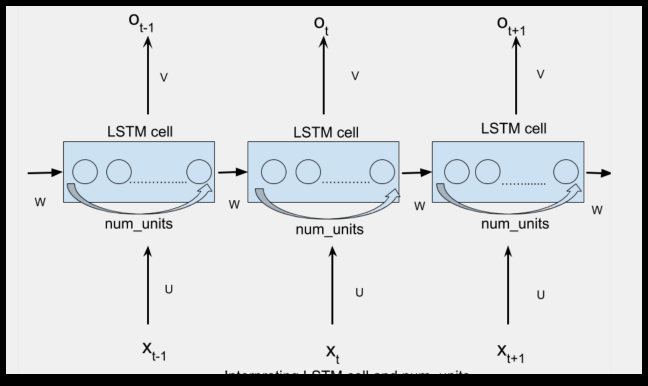


Application	Cell	Layers	Size	Vocabulary	Embedding Size	Learning Rate	
Speech Recognition (large vocabulary)	LSTM	5, 7	600, 1000	82K, 500K	--	--	paper
Speech Recognition	LSTM	1, 3, 5	250	--	--	0.001	paper
Machine Translation (seq2seq)	LSTM	4	1000	Source: 160K, Target: 80K	1,000	--	paper
Image Captioning	LSTM	--	512	--	512	(fixed)	paper
Image Generation	LSTM	--	256, 400, 800	--	--	--	paper
Question Answering	LSTM	2	500	--	300	--	pdf
Text Summarization	GRU		200	Source: 119K, Target: 68K	100	0.001	pdf

Chapter 4.3 - Implementation of LSTMs & RNNs

- Two challenges:
  - Preprocess the sequential data into numerical data
  - Represent memory in code
- nn.RNN(input_size, hidden_dim, n_layers, batch_first=True)
  - hidden_dim -> how many features the output of Hidden layer / RNN will have
  - n_layers -> how many hidden layers (normally 1-3
  - batch_first -> Batch_size will be the first dimension
- nn.Linear(hidden_dim, output_size)
  - final layer connecting the recurrent part to the output
- r_out, hidden = self.rnn(x, hidden)
  - Two outputs from the rnn! -> The r_out can then be passed into the final layer and the hidden used for the next RNN; return both in the forward function
- MSELoss as criterion, as we do not classify but have a regression loss
- Adam optimizer is the standard for RNNs
- Training loop: Initalize hidden state to None in the beginning
- x = Data[:-1] -> Everything except for the last
- y = Data[1:] -> Everything except for the first (since we always start with an x)
- We set the hidden output from the RNN to the hidden state -> Then it will be reused in the next loop
  - By detaching the hidden state from its history, we do not need to backpropagate through all past states
- The Batch Size is the number of sequences you use
  - You could pass in the range [1, 2, 3, 4, 5] as one sequence ; Batch Size of 1
  - Or you split it in two sequences [1, 2, 3] [4, 5] ; Batch Size of 2
  - We want to run multiple sequences through our network at a time, hence one batch contains multiple sequences
  - Each minibatch has the dimensions batch size by sequence length (note mini batch and batch are not the same here!)
  - The first minibatch on the right is hence [[1,2,3] [7,8,9]] with batch size being 2 (vertical dimension) and sequence length being 3 (horizontal dimension)
- Stacking multiple LSTM Cells next to each other
  - N_Hidden (or num_units) is how many LSTM Cells we throw our values in at each point in

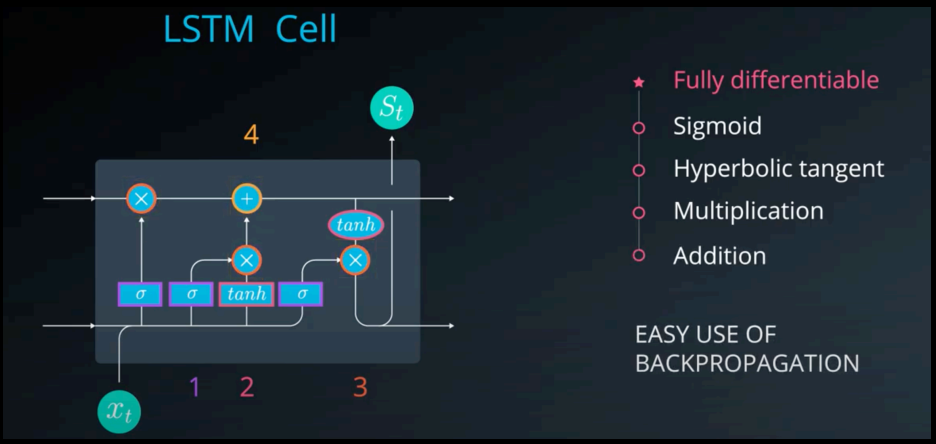




time; After those we then need a Linear layer to take N_Hidden as input and output how many classes via a softmax; or 1/0 via sigmoid
    - Larger values allow a net to learn more text features
- Input_Size is simply how many features each char or each x has. Remember, we pass in one char at a time for text sequences, and if we hot_encode this char it just as many features as hot_encode values (normally ~all values in the alphabet)
- Introducing Batch_sizes is not necessary (e.g. you can go with 1), but it allows for better parallelization; As we are just learning the weights, it doesn't matter if we mix in some input from the middle and the beginning
    - The actual input to a LSTM is of format: (seq_len, batch, input_size) - where input_size is also the PARAMETER of the LSTM
- The hidden state is what represents the memory of our LSTM

Chapter 4.2 - LSTMs

- Replacing the hidden neurons or state with the following cell:
- Sigmoids can decide how much data flows through (0 means none, 1 means all)
- At every stage ST Mem & LT Mem come in and ST & LT come out (the two arrows in and out)
- 4 different gates:
    - Input Modulation (/ Forget)
    - Learn
    - Remember
    - Use



- IM Gate: LTM comes in and partly gets deleted
- LE Gate: STM & X come in
- RE Gate: All three are joined and turned into the next LTM
- US Gate: All three, uses them and turns into STM (BOTH the prediction & the STM)
- LE GATE: X w/ circle means gets mutliplied elemnt-wise with a ignore scalar -> We get the ignore scalar via a sigmoid function (where we take x & STM times a weight) ;; It's the X circle below the + in the picture
- IM Gate: Again we take LTM times a forget vector which is a result of a sigmoid of X(t) & STM
- RE Gate: The + w/ circle adding the remainder of LTM & The STM & Input remainders
- US Gate: Take the RE input and apply a tanh & an ignore -> Output to St & STM

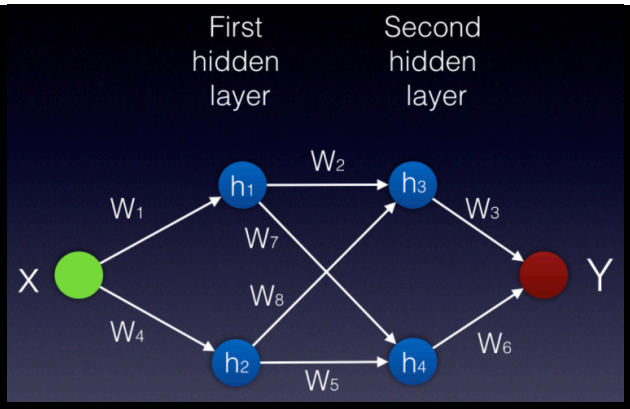
- You can change the activation functions & change the structure!
- -> Gated Recurrent Unit with only one working memory
- Peephole connection: Also connecting the LTM to the ignore scalar (the sigmoid values)
- http://blog.echen.me/2017/05/30/exploring-lstms/

Chapter 4.1 - RNNs

- Biological Neurons also have recurrent connections
- When input comes in over time – Video, Speech, Text
- History
  - TDNN – Time steps for RNN inputs X Limited to the time chosen
  - Elman Networks X Vanishing Grad Problem
- -> LSTMs
  - Fixed state variables which can be reintroduced later
- Applications:
  - Speech Recognition: Alexa, Dragon, Siri
  - Time Series Prediction:
    - Traffic Patterns – waze
    - Movie Selection – Netflix
    - Stock Movements – Quantitative Hedge Funds
  - NLP:
    - Machine Translation – Google; Salesforce
    - Question Answering – Google Analytics
    - Chatbots – Baidu, Slack, Google
  - Gesture Recognition: Qualcomm; eyeSight
- FFNNs (Feed forward) – Two Types
  - Classification (Usually Cross Entropy Error)
  - Regression (Usually MSE)
  - Easily calculate the total calculations in your network (X inputs * Neurons in next layer + Neurons in next layer * Neurons in layer after + …)
  - If you think of a graph, when it slopes up, substracting the weight gets you closer to the min; when it slopes down substracting the slope (which is negative in the case) also gets you close to the minimum

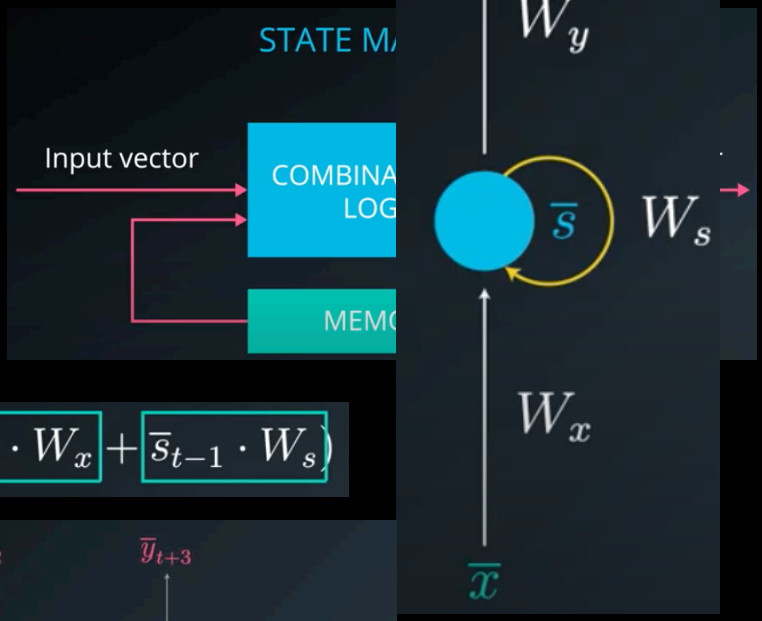
Equation A

$$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial h_3}\frac{\partial h_3}{\partial h_1}\frac{\partial h_1}{\partial W_1} + \frac{\partial y}{\partial h_4}\frac{\partial h_4}{\partial h_1}\frac{\partial h_1}{\partial W_1}$$

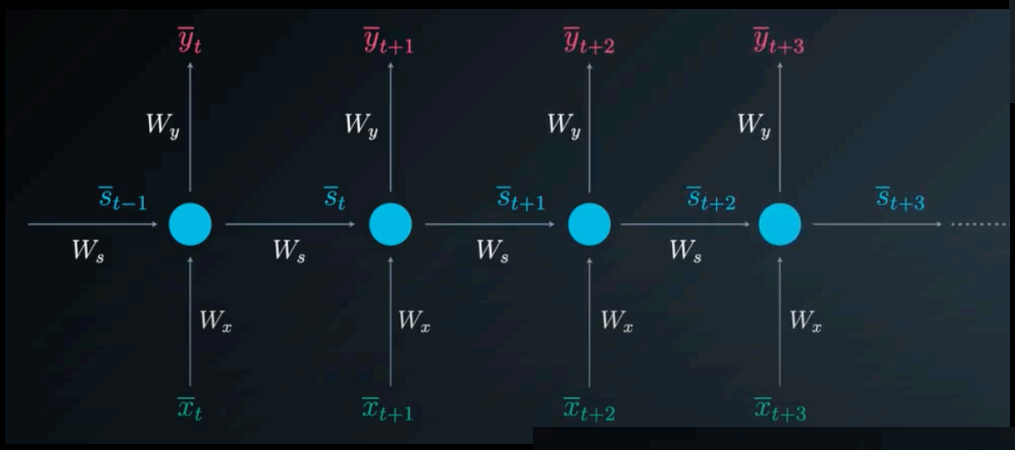


- Greek letter nabla to denote a vector with all partial derivatives of a function
- To the right PICTURE w/ example on updating a Weight in the beg.

- Two diffs btw RNNs & FFNNs:
  - We train with sequences as previous inputs matter
  - Outputs of hidden layer serve as memory (Instead of H for out output of a hidden layer, we refer to it as the state S)
- Similar to a state machine
- W(s) connects the previous state to the current state (ergo some weight attached to the previous state) — as outlined in PICTURE on the right called the

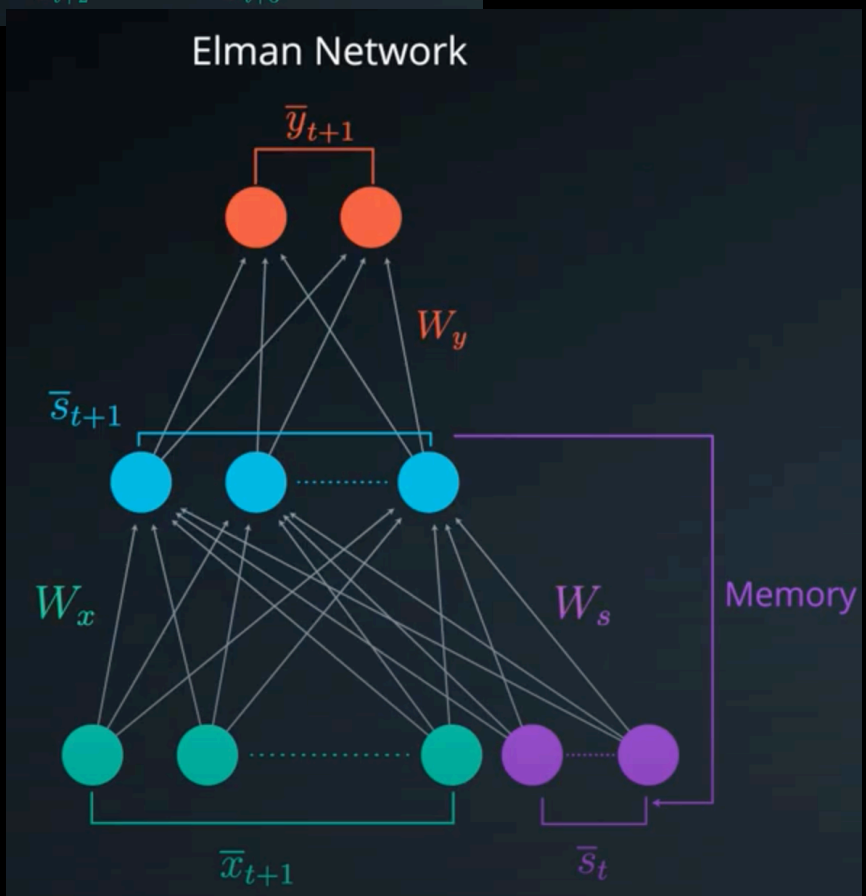


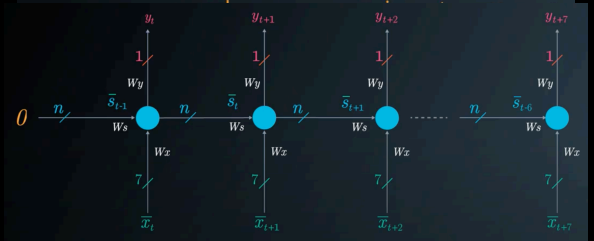
$$\overline{s}_t = \phi(\overline{x}_t \cdot W_x + \overline{s}_{t-1} \cdot W_s)$$



folded model; The other PICTURE outlines the same model unfolded
-We can have many outputs (e.g. time series prediction) or just one (e.g. sentiment analysis)

- Placing the states as seperate inputs to the hidden nodes we get the following representation of the Elman Network (PICTURE)
- RNN Example for word detection
- Yet Vanishing Grad Problem with RNN -> Gradual Adoption of LSTMs (RNNs can only look





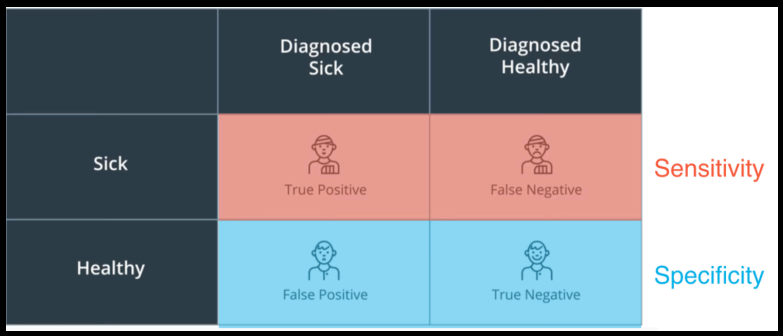
at past 10 steps while LSTM at past 1000

Chapter 3.9 - Git, Jobs, Ressources

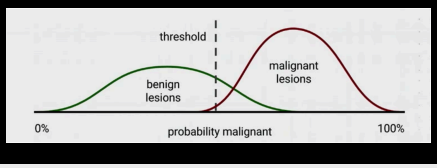
- Loads of Twitter/Medium/Blogs – Give them a try, stick only if the provide value (most don't)
- AWS DeepLens camera ; Google Vision Kit
  - https://aws.amazon.com/deeplens/
  - https://blog.google/technology/ai/introducing-aiy-vision-kit-make-devices-see/
- Common job titles in DL:
  - Machine Learning Engineer
  - Deep Learning Engineer
  - Machine Learning Researcher, Deep learning
  - Applied Research Scientist
  - Software Engineer, Machine Learning
  - Data Scientist
  - Boolean Searches on LinkedIn to find sth suited to your skills
  - https://www.linkedin.com/help/linkedin/answer/75814/using-boolean-search-on-linkedin?lang=en

Chapter 3.8 - Skin Cancer

- Sensitivity = TP / (TP+FN)
- Specificity = TN / (FP + TN)
- Recall = TP / (TP+FN)
- Precision = TP / (TP + FP)
- Lower threshold (e.g. 0.2 probability -> cancer) is better to minimize probability of sending someone home who has cancer; This does not matter for training the algorithm (as he will just try

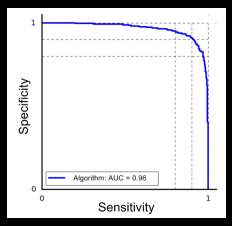


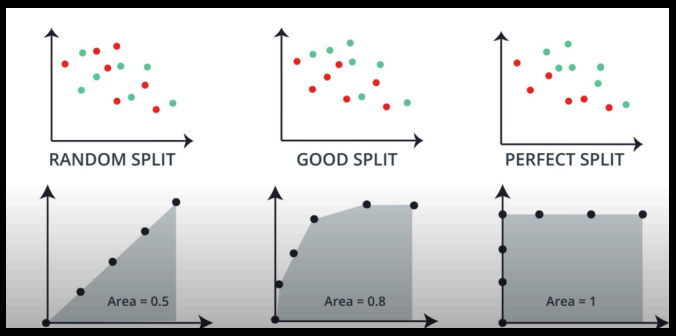
to maximize the probabilities towards 0 or 1), but it matters for testing / validation, when we cannot say 0.8 but 1 or 0



- Then based on our models outputs we decide for the threshold we want to set (e.g. 0.2 to make sure all patients are save)

- ROC Curves (Receiver Operating Characteristic or Sensitivity Specificity curve)
  - Determining all possibilities to split two different types of data and plotting the respective True Positive Rate (TP / All positives) and False Positive Rate (FP / All Negatives)





  - Three possible shape scenarios (Area = 0 is also possible, when e.g. there are only blue labels)
  - If we flip the ROC curve horizontally we get Specificity vs Sensitivity
- Confusion Matrix: Evaluating a model by adding the probabilities to the Sens/Spec table
  - False Positive: Type 1 Error
  - False Negative: Type 2 Error
  - Perfect matrix is the identity matrix (1's along the diagonal; means everybody has been diagnosed correctly in a multi class confusion matrix)

Chapter 3.7 - Project Dog

- CenterCropping might remove large parts of the image if the resolution is higher than the center crop -> Better to use Resize
- Transforms.ToTensor turns the images pixel values for each channel in a range from 0 to 1 (by turning it into a torch tensor first and then dividing by 255)
  - Normalizing then means that we try to group those values around the same center; obv the average mean might be considered to be 0.5 & 0.5 std, but since images tend to be slightly darker and closer together bopth turn out to be lower
- [0.485, 0.456, 0.406] **is** the normalized mean value of ImageNet, and [0.229, 0.224, 0.225] denotes the std of ImageNet.
- Dropout before linear layers (e.g. drops out p percent of the x values coming into that layer)
- Compress folder into zip: tar -c -z -f myarchive.tar.gz <Folder_name>

-

Chapter 3.6 - Style Transfer

- Content image & Style Image ->
  - C Img: Takes its content – mainly object & shape arrangement
    - We take the ocntent from the later conv layers
  - S Img: Takes its style – mainly colors & textures
    - We take the style from diff conv layers
- -> Content Loss & Style Loss
  - CL: Loss function for between Content of orig. image and Content of created image
  - SL: Gram Matrix to represent style
    - Flatten x & y dimension of feature map into vector (stacking up all feature maps we get a matrix form of the conv layer)
    - We take this matrix's transpose and multiply it by the matrix to get the gram matrix
      - The advantage here is that now each value contains non localized information (but rather color relationships
      - Each value in the gram matrix will indicate the similarity between one row and one column (Since we use the sum of the multiploied values of a row & column in a matrix multiploication)
      - There are alsoi other alternaties to representing style apart from the gram matrix
    - Loss: Mean Sqaured Distance between Gram matrices for each Conv Layer with a weight for each conv layer (Ts is here a list of all the target conv layers gram matrices)

$$\mathcal{L}_{style} = a \sum_i w_i (T_{s,i} - S_{s,i})^2$$

- (A Conv Layer with depth of 8 has 8 feature maps)
- Add up both losses for total loss (To ensure both are weighted equally in the total loss add alpha and beta weights) -> alpha over beta as the content(alpha) vs style(beta) ratio – The higher the ratio the more weight is given to content
- To get a representation of the content we choose a very deep layer in our conv. NN to have it striped off most of the style

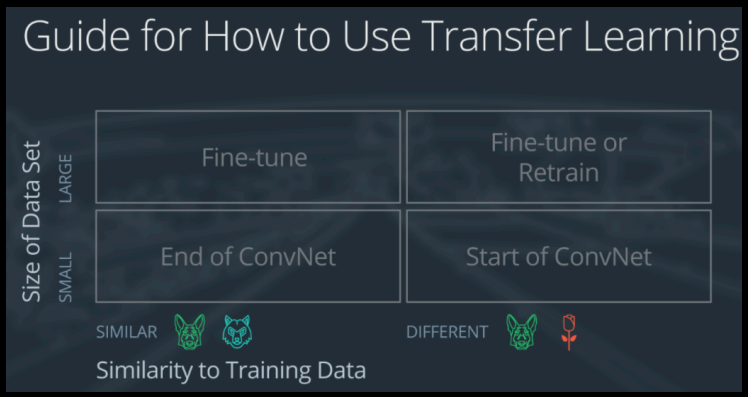
Chapter 3.5 - Autoencoder

- Picture -> Encoder (Sim. to conv layers) -> Flatten -> Decoder -> Picture
- Use cases
  - Data compression

    - similar to jpeg file, but while a jpeg file has predefined compression function an autoencoder learns them himself (yet no as efficient as jpeg)
  - De-noising images
  - Image transformation (e.g. from grayscale to color)
  - Get training loss as low as possible
- For pictures – use conv layers, specifically transpose conv layers
- 2 ways to encode: Upsampling or Transpose convolution
  - Transpose convolution works via combining the input values across different directions with a padding of zeros (it hence generates values based on what its given)
- MSELoss (Mean squared error), since it is a regression task and we want to compare QUANTITIES (e.g 10 - 8) ; Cross Entropy is better for comparing probabilistic values
- It seems more effective to only use activation functions on the decoder side (my solution vs udacity solution; i get 14.5 vs uda 16.5 loss)
  - Problems with curved areas as Transpose convolution works in a grid like pattern
  - -> Nearest Neighbor Interpolation combined with normal convolutional layers
  - Nearest Neighbor Interpolation is an upsampling method copying the pixels from a 2x2 to a e.g. 4x4 area
    - F.upsample (/F.interpolate)

Chapter 3.3 & 3.4 - Transfer Learning, Weight Initialization

- First Conv Layer learns e.g. edges or blocks of color
  - 2nd Circles, stripes, rectangles
  - These early layer features are useful for every kind of images
  - The final layers though will be very specific, such as for a model trained on birds maybe specific type of birds
  - -> For Transfer learning it may be helpful to remove the final layers
  - There must be some sort of randomness in the wight initialization to give the backprop the chance to differentiate and adjust weights
  - The weights should be inversely related to the number of inputs a node sees to keep the values small
    - in the range of - 1 / sq.root(no of inputs) to + 1 / sq.root(no of inputs)
      - no of inputs often denoted as n
      - With this method they will be centered around zero and inversely related to n
    - uniform distribution (taking equally random values in a range) vs normal distributrion (taking values centered around mean; normally 0)



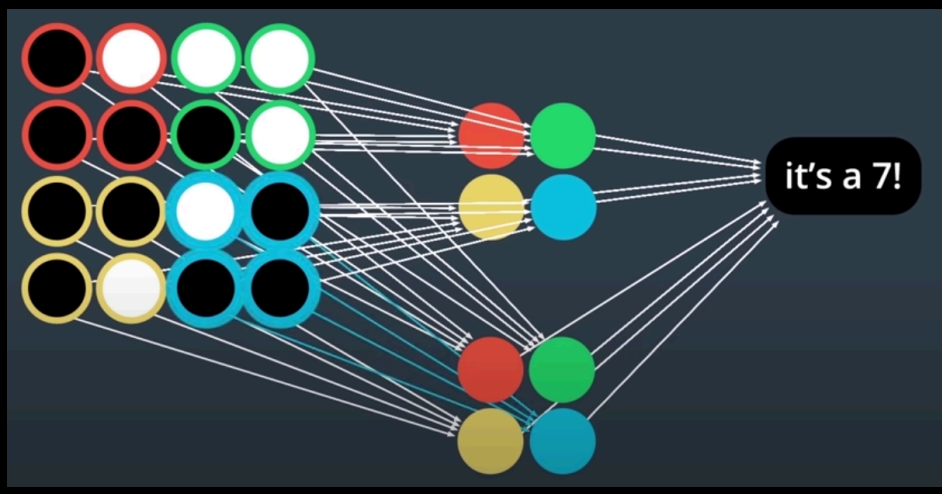
Chapter 3.2 - AWS

- AWS has EC2s – Elastic Compute Clouds allowing you to launch an ‘instance’ (a virtual server) – You can attach GPUs to these
- p2x.large for Udacity
  - P2 Instance is GPU enabled
- AMI – Amazon Machine Image ; Template w/ a software configuration (e.g. an operating system
  - From this AMI you launch an instance
  - -> Deep Learning AMI Linux
- Cloud: Geographically distributed data centers hosting Virtual Machines (VMs) of different users
- Shut down EC2 Instance when taking a break or else (it consumers credits!)
- Amazon SageMaker: ML Platform to build train & test models quicker & easier
- 2 Advantages of Clouds:
  - Better computing power
  - Faster Deployment to the internet
- Charges by AWS:
  - Running instances
  - Storing charges (Only disappear if you DELETE the instance)
  - You cannot limit AWS from charging you at a certain point, but you can set alerts
  - https://aws.amazon.com/ec2/pricing/on-demand/

Chapter 3 - CNN’s

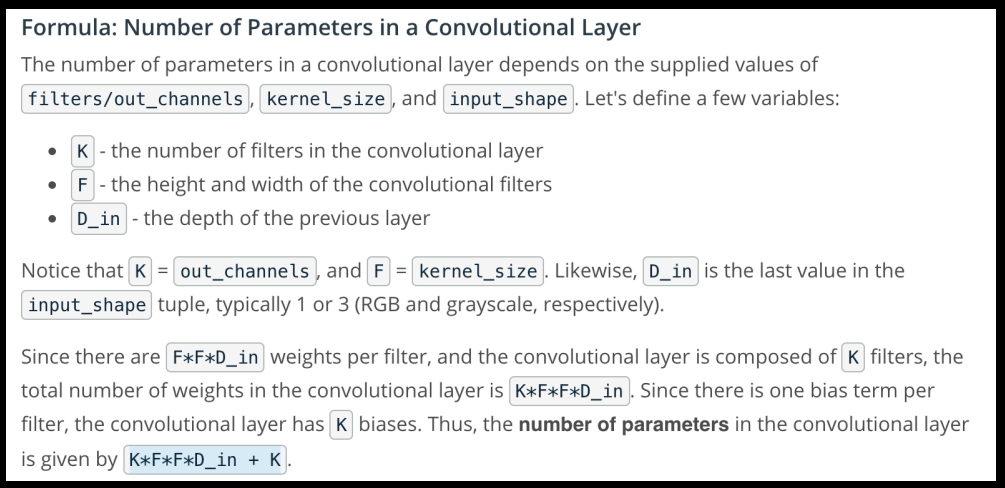
- Flattening – the process of turning an image into a vector
- Normalizing:
  - After you have divided the RGB values by 255 and brought in 0 - 1 range, we substract the mean (centers the data around zero) and then divide by the Standard D - crops the data closer to each other ;;; For easiness we often approximate both to 0.5, but better to calculate the actual values!
- To get an idea for how many hidden layers, try a G search on the dataset / type of images (e..g small greyscale images)
- The main advantage of our vision vs computers lies in the object detection part – we simply get far more images (many thousands every second of our life) and have a good memory system
- Cross_Entropy first applies softmax and then NLLLoss -> Instead apply softmax in your forward loop and then use only NLLLoss as cost function
- FC for fully connected
- Pipeline:
  - Load Data
  - Visualize Data to get a feeling
  - Preprocess (Normalize, Transform)
  - Define a model – Do your research – has somebody done the same e.g. on github?
  - Train your model
  - Save the best model via validation test
  - Test!
- From MLP (Multilayer Perceptron) to CNNs

- MLP: Flatten to a vector
  - Huge amount of parameters for a fully connected network
  - Loose the ‘2D’ features
- CNN:
  - Sparsely connected layers
    - Don’t feed every hidden node every input -> Each hidden node only understands a part of the image
    -
  - Take full 2D matrix as input



- High frequency: complex pattern / High chage in brightness -> Edges of an image!
- Low frequency: monotone pattern
- Applied to filters:
  - High pass filters -
    - enhance high frequency parts of an image to sharpen it
    - edge detection via kernel matrices
      - All values in a kernel must add to zero to not brighten / darken the total image
      - Kernel convolution operatiom represented via an asterisk#
- Three ways to deal with border pixels:
  - Extend (extend border by one field for calculation when needed)
  - Padding (Pad image with a border of 0’s (black pixels))
  - Crop (Any pixel at the edges gets skipped resulting in a slightly smaller image)
- When we apply different filters to an image we may apply e.g. one kernel to detect horizontal edges and one kernel to detect vertical ones and get two image outputs
- Color images are 3D tensors; Gray images 2D
  - The depth of an RGB image (the ‘third D’) is 3
- Convolutional layers are not dense layers (Fully connected to all nodes in prev layer)
- HYPM 1: No of nodes in a CNN layer corresponds to number of filters
- HYPM 2: To increase the amount of patterns it detects, increase the ‘size of the filter’
- HYPM 3: Stride – how much each kernel moves (1 by default)
  - If stride is 2 -> The ConvLayer will only be half the width & height of the image (slightly diff depending on the edges)
  - Edge dealing:
    - Crop – We loose some information (no information about several edge parts - may be very big if big stride)
- Pooling Layers to reduce the image size and make it more computationally efficient
  - Max Pooling, Avg Poolig
  - Problem: Discards some features
- Capsule Network – recognizing spatial relationships (e.g. only a face if 2 eyes, 1 mouth..)
  - Paper from G Hinton:
  - https://video.udacity-data.com/topher/2018/November/5bfdca4f_dynamic-routing/dynamic-routing.pdf
  - https://cezannec.github.io/Capsule_Networks/
  - Parent & Child Nodes to make up a complete picture (e.g. face is parent -> mouth is a child)
  - Two outputs of a capsule:
    - m – Magnitude: Probabilty as to whether a part exists

    - theta – Orientation: State of the part properties
  - Designed after how humans focus – We focus on a small part of a scene and build up the complete picture piece by piece
    - When we look aat sth - we only focus our eyes at a single point in time
- Resizing images – normally to a square
- Defining a Conv Layer:
  - self.conv1 = nn.Conv2D (3, 16, 3, stride = 1, padding = 1
  - 3: Size of input depth - in this case 3 for RGB
  - 16: Size of output depth - 16 for 16 diff filtered images
  - 3: Size of kernel – in this case 3x3
  - stride = 1: How the kernel moves
  - padding = 1: How much padding to ensure same output size -> Depends on your kernel size & Stride! -> For stride 1 & 3x3 1 is good
    - https://cs231n.github.io/convolutional-networks/#conv
- We generally want to increase the depth with each Conv to extract more features
- Pooling layers after each or every 2nd Conv Layer
- Defining a Max Pool Layer:
  - -self.maxpool = nn.MaxPool2d(2, 2)
    - 2 & 2: kernel size & stride -> 2 will halve it from previous layer
- After Conv Layers we flatten the image into a feature vector
- Invariant representation: Independent of angle (Rotation invariance) & Independent of Scale (Scale Invariance) & Independent of whether on the left or right (Translation Invariance)
  - We get some translation invariance from Max Pooling layers
  - We can fight all of them by augmenting our data



Formula: Number of Parameters in a Convolutional Layer

The number of parameters in a convolutional layer depends on the supplied values of `filters/out_channels`, `kernel_size`, and `input_shape`. Let's define a few variables:

- `K` - the number of filters in the convolutional layer
- `F` - the height and width of the convolutional filters
- `D_in` - the depth of the previous layer

Notice that `K` = `out_channels`, and `F` = `kernel_size`. Likewise, `D_in` is the last value in the `input_shape` tuple, typically 1 or 3 (RGB and grayscale, respectively).

Since there are `F*F*D_in` weights per filter, and the convolutional layer is composed of `K` filters, the total number of weights in the convolutional layer is `K*F*F*D_in`. Since there is one bias term per filter, the convolutional layer has `K` biases. Thus, the **number of parameters** in the convolutional layer is given by `K*F*F*D_in + K`.

- L2 Regularization penalizes a vector such as (0.5,0.5) as 0.5 (sum of squares), while L1 penalizes it with 1 -> L1 tends to diverge towards zeroing out some weights by keeping the vectors at e.g. 1,0, which has the same cost as 0.5,0.5 for L1
- If problems with finding global minimu (even though unlikely acc to Andrew due to high dimensionality) random restart could be a solution
- Why do we use sigmoid derived grad descent despite not using sigmoid fnctn
- Rather than tweaking too many hyperparameters, the BIGGEST node is the data - the NN is just the tool
  - Reducing noise – making it more obvious for the NN what is important

- **PyTorch**
- weights = torch.randn_like(features)
  - With the same metrics as for features
- .reshape vs .resize_ vs view
  - view creates new matrix with given shape; resize modifies same matrix; .reshape does one or the other
- Instead of torch.mm, you can always use torch.multiply and then torch.sum (same for numpy)
- torch.from_numpy to turn np array in a tensor
- a.numpy() turns torch array in numpy
  - Both work on one same space in memory
- class Network (nn.Module):
  - Inherit form Pytorchs nn.Module class to get useful methods
  - After init must put in the next line super().__init__()
- For softmax function pass the values pre-softmax into nn.CrossEntropyLoss
  - This raw output is called 'logits/scores'
- next() returns next item in an interator
- optimizer.zero_grad() to clear the gradients from previous gradients
- When saving a state_dict and then loading it again into a model, the model must have the same parameters! (same shapes)
  - Define params in checkpoint file
- Transforms.compose by itself does not augment the data – it returns just as much images as it got – FiveCrop can augment it

```
model = nn.Sequential(nn.Linear(784, 128),
                      nn.ReLU(),
                      nn.Linear(128, 64),
                      nn.ReLU(),
                      nn.Linear(64, 10),
                      nn.LogSoftmax(dim=1))

criterion = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.003)

epochs = 5
for e in range(epochs):
    running_loss = 0
    for images, labels in trainloader:
        # Flatten MNIST images into a 784 long vector
        images = images.view(images.shape[0], -1)

        # TODO: Training pass
        optimizer.zero_grad()

        output = model.forward(images)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
    else:
        print(f"Training loss: {running_loss/len(trainloader)}")
```