Lecture 2

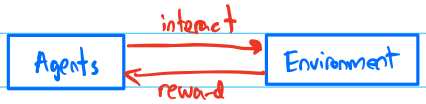
Unsupervised Learning

- No label or target class
- Find out properties of the structure of the data → Autoencoder
- Clustering

Supervised Learning

- Labels or target classes
- Learn from train data → Learning = how well the model do on the test data

Reinforcement Learning



Reward agent for desired behavior and/or punish undesired one.

Image Classification → Supervised Learning

Nearest Neighbor → Supervised Learning

↳ Which of the other images in the set is the closest to this one (compute distance)

K - Nearest Neighbor → take k nearest neighbor and classifier image based on majority

↳ higher k can give less variance

Split data:
- train data → 60% / 90% → train with train data
- validation data → 20% / 5% → improve performance with the validation data
- test data → 20% / 5% → check final performance with test data

Cross-validation

Split data into n-fold (random k fold for validation)

↳ train data with another n-k fold

↳ use k validation test set to optimize hyperparameter

↳ Train data and test again for the next split/run

↳ Do it n runs/split, each time each validation fold

Then pick the hyperparameter with the least average loss

Linear Regression → Supervised Learning

$$\hat{y}_i = \theta_0 + \sum_{j=1}^{n} x_{ij}\theta_j$$

bias weight; j: no of feature (column); x_{ij}: input, feature; θ_j: weight

i: i-th sample that we predict

x_i = i-th row of x

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_n \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1d} \\ \vdots & \vdots & & \vdots \\ 1 & x_{1n} & & x_{nd} \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_d \end{bmatrix}$$

bias

Loss function (Optimization)

① Linear Least Square $\min_{\theta} J(\theta) = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2$ $\theta = (X^T X)^{-1} X^T y$

② Maximum Likelihood Estimate → $\theta = \max \log p(y|x,\theta)$
↳ based on probability distribution

if y is normal distribution and $y_i = x_i \theta$ → $\theta = (X^T X)^{-1} X^T y$

$p(y_i) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{1}{2\sigma^2}(y_i - x_i\theta)^2}$ $y_i \sim N(x_i\theta, \sigma^2)$

Regression → predict a continuous value (temp. of a room) → We can transform regression output to binary output by interpreting output of regression as probability distribution ($P > 50\%$ - Class A $P \leq 50\%$ - Class B)

Classification → Predict a discrete value
↳ Binary → predict either 0 or 1
↳ Multiclass → Predict set of N classes

Logistic Regression

Estimate probability of an event, bounded between 0 and 1
Loss function with $p(y)$ = Bernoulli trial and activation function σ sigmoid

Binary Cross-Entropy Loss (BCE) = $L(\hat{y}_i, y_i) = y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)$ → Loss of all training

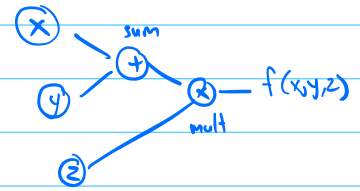
Cost function $= -\frac{1}{n} \sum_{i=1}^{n} L(\hat{y}_i, y_i)$ → Average loss over all training

Lecture 3

Computation Graphs

Forward pass

$f(x,y,z) = z(x+y)$



Regression Loss

$$\text{L1 Loss}: \quad L(y,\hat{y},\theta) = \frac{1}{n}\sum_{i=1}^{n} \|\hat{y}_i - y_i\|_1$$

$$\text{MSE Loss}: \quad L(y,\hat{y},\theta) = \frac{1}{n}\sum_{i=1}^{n} \|\hat{y}_i - y_i\|_2^2$$

Binary Cross Entropy → for binary classification

$$L(y,\hat{y};\theta) = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)$$

For multiple class **Cross Entropy Loss**

$$L(y,\hat{y};\theta) = \frac{1}{N}\sum_{i=1}^{n}\sum_{k=1}^{C} -y_{ik}\log\hat{y}_{ik}$$

(C = # of class)

↖ this generalise binary case → $y_{ik} = 1$ if the true label (class) of the i-th sample is the same as class k. 0 otherwise (one-hot encoding)

Minimize the loss → find better prediction

Minimize Loss with respect to θ

$$\theta = \theta - \alpha \nabla L(y,\hat{y})$$

(α = learning rate) → find optimum θ that result in minimum loss → Gradient Descent

Lecture 4 - Optimization and Backpropagation

Neural Network

5 layers → $f = W_5 \sigma\left(W_4 \tanh\left(W_3, \max\left(0, W_2 \max(0, W_1 x)\right)\right)\right)$

# of depth of NN = # of hidden + output layer

# of node in input layer = # of column/feature in data

ReLU is the most used Activation Function

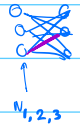
Regression Loss → L1 Loss → $\sum_{i=1}^{n} \frac{1}{n} \| y_i - \hat{y}_i \|_1$

→ MSE Loss → $\sum_{i=1}^{n} \frac{1}{n} \| y_i - \hat{y}_i \|_2^2$

Classification loss → Cross Entropy Loss = $\frac{1}{N} \sum_{i=1}^{n} \sum_{k=1}^{c} -y_{ik} \log \hat{y}_{ik}$

Back propagation $\nabla_\theta L(\theta)$

x_k - input variables

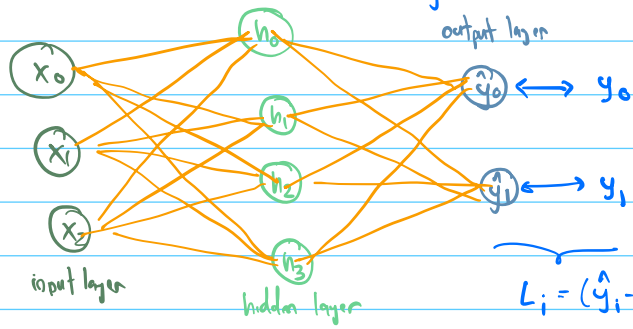


$w_{k,m,n}$ - weight of n-th neuron in k-th layer to m-th neuron in the next layer

$\hat{y}_i$ - output of i-th row y_i - ground truth targets L - loss function

forward ⇒ $f(x, w) = \tanh(w_2 \underbrace{\sigma(x w_1)}_{s}) \Rightarrow L(y, f(x,w)) = CE(\tanh \_\_\_\_)$

backward ⇒ for each weight w_i ⇒ for $w_2 \rightarrow \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_2}$ for $w_1 \rightarrow \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial s} \cdot \frac{\partial s}{\partial w_1}$



$L_i = (\hat{y}_i - y_i)^2$ or $L_i = y_i \log(\hat{y}_i)$

$y_i = A\left(b_{1,j,i} + \sum_j h_j w_{1,j,i,j}\right)$

↳ let $A(x) = \max(0, x)$

$\frac{\partial L}{\partial w_{1,j,i,j}} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial w_{1,j,i,j}} = 2(\hat{y}_i - y_i) \cdot \frac{\partial \hat{y}_i}{\partial w_{1,j,i,j}}$

$= 2(\hat{y}_i - y_i) \cdot h_j$

↳ to $\partial \hat{y}_i$ use $\max(0,x)$ $\frac{\partial \hat{y}_i}{\partial w_{1,j,i,j}} = \begin{cases} 0 \text{ if } x \le 0 \\ h_j \text{ if } x > 0 \end{cases}$

Calculate # of weight in each layer

Layer = (# of neurons · # input neuron) + # of biases (usually just # of neuron in layer)

(Input layer doesn't have any weight, it start in 1st hidden layer)

if A = Sigmoid $= \frac{1}{1+e^{-s}}$ and $L = -\sum_{i=1}^{n_{output}} [y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)] \Rightarrow$ BCE

$\searrow s = \sum_{i=1}^{n} h_j w_{ji}$

$\frac{\partial L}{\partial w_{ji}} = \frac{\partial L}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial s} \cdot \frac{\partial s}{\partial w_{ji}}$

$= \left(-\frac{y_i}{\hat{y}_i} + \frac{1-y_i}{1-\hat{y}_i}\right) \cdot \left(\hat{y}_i(1-\hat{y}_i)\right) \cdot (h_j)$

$-\frac{y_i + \hat{y}_i}{\hat{y}_i(1-\hat{y}_i)} \cdot (\hat{y}_i(1-\hat{y}_i))(h_j)$

$= (-y_i + \hat{y}_i)h_j$

$\frac{\partial L}{\partial s} = -y_i(1-\hat{y}_i) + \hat{y}_i(1-y_i)$

$= \hat{y}_i - y_i$

Regularization

→ Use to reduce overfitting → Aim to lower validation error and increase training error

Loss function $= \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 + \lambda R(\theta) \Rightarrow \lambda R(\theta)$

↳ some constant λ

↳ if λ is large, we care more about regularization

↳ if λ is small, care more about predicting as close as possible

Regularization technique

- → L2 Regularization
- → L1 Regularization
- → Max Norm Regularization
- → Dropout (Drop some of the neuron)
- → Early stopping

$\theta_1 = [0, 0.75, 0]$

$\theta_2 = [0.25, 0.5, 0.25]$

L2 Regularization → $R(\theta) = \sum_{i=1}^{n} \theta_i^2$ → penalize larger weight → Want weight to have similar values → Take all feature into account

$R(\theta_{11}, \theta_{12}, \theta_{13}) = 0^2 + 0.075^2 + 0^2 = 0.5625$ $R(\theta_{21}, \theta_{22}, \theta_{23}) = 0.25^2 + 0.5^2 + 0.25^2 = 0.375$ → Use this one

L1 - Regularization → $R(\theta) = \sum_{i=1}^{n} |\theta_i|$ → Enforce Sparsity → will focus on a few key feature that have high θ

$R(\theta_{11}, \theta_{12}, \theta_{13}) = 0 + 0.75 + 0 = 0.75$ $R(\theta_{21}, \theta_{22}, \theta_{23}) = 0.25 + 0.5 + 0.25 = 1$

↳ use this one

Deep NN (multiple layer)

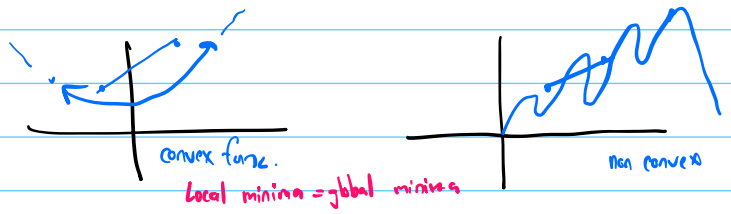
Adv → better at generalization

Problem → might cause vanishing gradient, memory

Lecture 5 - Scaling Optimization

Gradient Descent

- Gradient steps in direction of negative descent $x' = x - \alpha \nabla_x f(x)$
- Not guarantee to reach global optimum, only local minimum
- NN is a non-convex function with many local minima



- Learning Rate → Big Learning Rate might cause weight to bounce off local minima
  → Small Learning Rate might cause learning to be low esp. if it's in plateau part

$$\theta^{k+1} = \theta^k - \alpha \nabla_\theta L_i(\theta^k, x_i, y_i)$$

Iterate until convergence $|\theta^{k+1} - \theta^k| < \varepsilon$

$$L = \frac{1}{n} \sum_{i=1}^{n} L_i(\theta, x_i, y_i) \quad \text{where} \quad \theta = \arg\min L$$

For Multiple Training sample

→ the gradient is average of all data (or batch)

$$\nabla_\theta L = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta L(\theta^k, x_i, y_i)$$

(n: no. of data)

For big data, Gradient Descent might take a very long time to update weight since it only update weight after 1 epoch

Stochastic Gradient Descent (SGD)

- We divide data into mini-batch and then iterate through each minibatch.

Each time it finish the minibatch, calculate the gradient and update the weight

- We can express total loss over the training data as expectation of all samples.
The expectation can be approximate with a small subset of data

$$\mathbb{E}_{i\sim[1,\dots,n]}[L_i(\theta, x_i, y_i)] \approx \frac{1}{|S|}\sum_{j\in S} L_j(\theta, x_j, y_j) \quad \text{with } S \text{ as set of minibatch}$$

- if there's n data and each minibatch has m data, then #of minibatch $\propto \frac{n}{m}$

- Smaller batch size means greater variance in the gradients → noisy update

For each iteration (batch)

1. Pick minibatch
2. Feed it to NN
3. Calculate the mean gradient of the minibatch
4. Update the weights with the mean gradient
5. Iterate step 1-4 for n minibatch → When we go through all minibatch, this is 1 epoch

Convergence of SGD

we want to minimize $F(\theta)$ (Loss for each minibatch)

$\mathbb{E}(H(\theta^k, x)) = \nabla F(\theta^k)$ $\quad \theta^{k+1} = \theta^k - \alpha_k \nabla_\theta H(\theta^k, x)$ where $\alpha_{1\dots n}$ is a sequence of positive step-sizes

This converge to local (global) minimum if this is all true

1. $\alpha_n \geq 0$, $\forall n \geq 0$
2. $\sum_{i=1}^{n} \alpha_n = \infty$
3. $\sum_{i=1}^{n} \alpha_n^2 < \infty$
4. $F(\theta)$ is strictly convex

Problem w/ SGD → it scales weight equally across all dimension (1 learning rate for all dim)

Gradient Descent with Momentum → Look at the direction of gradient (average gradient)

$$v^{k+1} = \beta v^k - \alpha \nabla_\theta L(\theta^k)$$

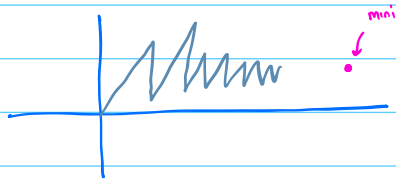
momentum, learning rate, gradient of minibatch

velocity (in vector-value) so direction important

$$\theta^{k+1} = \theta^k + v^{k+1}$$

current weight, updated weight, velocity

- If the gradient point in the same direct, build up velocity (v^{k+1}) and weight will change more
- If gradient keep changing direction, then we want to slow down (take smaller step)
  - ↳ v^{k+1} will stay the same cuz it get add and subtract



Move right faster since gradient go on the same direction
Move up and down slower since gradient change btw up and down alot

Hyperparameter = β, α

Root Mean Squared Prop (RMSProp) → look at the size of gradient (squared of average gradient)

→ Dampening direction with high variance (gradient) by divide learning rate by exponential average of squared gradients (size of gradient)

This mean high gradient will has less effect on changing of θ

This also mean that we can increase learning rate α and it will

1. speed up the direction with less variance/gradient (like momentum)
2. while not effect the direction of high variance/gradient as much like of dampening

$$\theta^{k+1} = \theta^k - \alpha \cdot \frac{\nabla_\theta L}{\sqrt{s^{k+1}} + \varepsilon}$$

where $s^{k+1} = \beta \cdot s^k + (1-\beta)[\nabla_\theta L \cdot \nabla_\theta L]$

s^{k+1}: exponential average sum of squared

$\nabla_\theta L^2 = \left(\sum_{i=1}^{n} \frac{1}{n} \frac{\partial L}{\partial \theta}\right)^2$ = size of gradient

Adam (Adaptive Moment Estimation)

Combine Momentum and RMSProp idea

where Momentum = θ increase faster when keep going in the same direction and θ stay still when keep going in opposite direction

and RMSProp = dampening direction with high gradient/variance which mean we can increase the α to help speed up

Combine together, Adam can

1. speed up when go in same direction
2. increase learning rate to speed up direction with less variance while not effecting direction with high variance too much.

first momentum: $m^{k+1} = \beta_1 m^k + (1-\beta_1)\nabla_\theta L(\theta^k) \rightarrow \hat{m}^{k+1} = \frac{m^{k+1}}{1-\beta_1^{k+1}}$

second momentum: $v^{k+1} = \beta_2 v^k + (1-\beta_2)\nabla_\theta L(\theta^k)^2 \rightarrow \hat{v}^{k+1} = \frac{v^{k+1}}{1-\beta_2^{k+1}}$

$\theta^{k+1} = \theta^k - \alpha\left(\frac{\hat{m}^{k+1}}{\sqrt{\hat{v}^{k+1}} + \varepsilon}\right)$

for bias toward 0 correction

β_2 to the power of k+1

Hyperparameter = $\beta_1, \beta_2, \alpha, \varepsilon$

Second Derivative

- Don't work well with minibatches but...
- Converge to local minima fast (in term of # of iteration)
- Use curvature of gradient to take a more direct route

Conclusion

Gradient Descent → calculate gradient of all data and adjust the weight

SGD → calculate gradient and adjust weight per batch

GD with Momentum → Gradient go in same direction make θ change faster or stay still if direction is fluctuated

RMSProp → Dampen change of gradient for the direction with high variance/gradient. So we can increase learning speed

Adam → change of θ faster if gradient go in the same direction and dampen gradient if it's high variance. So we can increase learning rate

GD → $\theta^{k+1} = \theta^k - \alpha\nabla_\theta L$ SGD → $\theta^{k+1} = \theta^k - \alpha\nabla_\theta L_{\text{mini-batch}}$ GD w/ Momentum → $v^{k+1} = \beta v^k - \alpha\nabla_\theta L$ $\theta^{k+1} = \theta^k + v^{k+1}$

RMSProp → $s^{k+1} = \beta s^k + (1-\beta)\nabla_\theta L^2$ $\theta^{k+1} = \theta^k - \alpha\frac{\nabla_\theta L}{\sqrt{s^{k+1}} + \varepsilon}$

Adam → $m^{k+1} = \beta_1 m^k + (1-\beta_1)\nabla_\theta L \rightarrow \hat{m}^{k+1} = \frac{m^{k+1}}{1-\beta_1^{k+1}}$

$v^{k+1} = \beta_2 v^k + (1-\beta_2)\nabla_\theta L^2 \rightarrow \hat{v}^{k+1} = \frac{v^{k+1}}{1-\beta_2^{k+1}}$

$\theta^{k+1} = \theta^k - \alpha\frac{\hat{m}^{k+1}}{\sqrt{\hat{v}^{k+1}} + \varepsilon}$

Lecture 6 - Training NN Part 1

Learning Rate Decay

Many option → 1. $\alpha = \frac{\alpha_0}{1 + (\#\text{ of epoch} \times t)}$

(t is decay rate)

→ 2. Step decay => $\alpha = \alpha - t\alpha$ every n epochs

→ 3. Exponential decay => $\alpha = t^{epoch} \cdot \alpha_0$

→ 4. $\alpha = \frac{t}{\sqrt{epoch}} \cdot \alpha_0$

Learning means generalization to unknown dataset → test optimize parameters w/ unknown dataset

(unknown dataset = validation data)

Training set → train NN

Validation set → optimizing hyperparameter, check generalization progress

Test set → test at the very end

if Training error —high→ have bigger model
train longer (underfitting)
diff architecture
diff hyperparameter / LR

↓ low

if Validation error (overfitting) —high→ More data
Regularization
New model architecture

↓ low

if diff btw training error and validation —high→ Make training data more similar to test data
Data augmentation
New model architecture

↓ low

Test error high —high→ More dev set data

↓

Done

Underfitting → training and validation loss/error can be further decrease at the end of training

Overfitting → training loss is decrease while validation stop decrease or increase (so it does not generalise well anymore)
↑ not as much

Bad sign if validation loss is less than training loss

Hyperparameter Tuning

1. Manual Search → You randomly select value for hyperparameter
2. Grid Search (Coarse) → Define a list or a range with step for all hyperparameter
   Then iteration over all value of hyperparameter
   So if there's 3 hyperparameters, 4 values each
   Then the number of iteration = 4^3
   After getting the best hyperparameter → Define a list or range around that value → Refine Grid Search
3. Random Search → Define a range for all hyperparameter and a number of iteration (combination) that you want to random.

How to start Training

- Start with Single training sample
  → check that model has capable of overfitting (accuracy should be 100%)
- Increase to handful of sample
- As you increase, model should generalise more

Fix Underfitting by – 1. Increase model's capacity → bigger model size → more layers/nodes/weight

Fix Overfitting by – 1. Reduce capacity → smaller model size → less layer/node/weight
– 2. Regularization → weight decay, Dropout, Batch Norm, L1/L2 Reg, Early Stop
– 3. Data augmentation → More training data

Lecture 7 - Training NN Part 2

Regression Losses

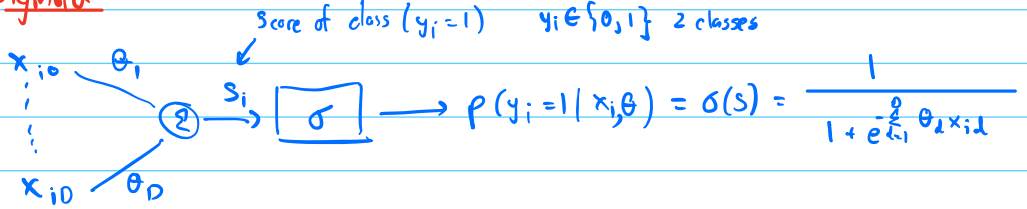
L2 Loss $\rightarrow L_2 = \sum_{i=1}^{n} |y_i - \hat{y}_i|^2$

- Sum of squared difference
- Prone to outlier
- Efficient to optimize
- Optimum = mean

L1 Losses $\rightarrow L_1 = \sum_{i=1}^{n} |y_i - \hat{y}_i|$

- sum of absolute diff
- Robust (cost of outlier is linear)
- Costly to optimize
- Optimum = median

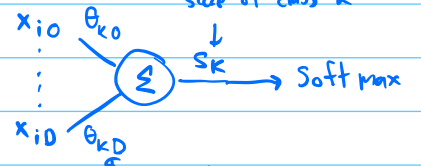
Sigmoid

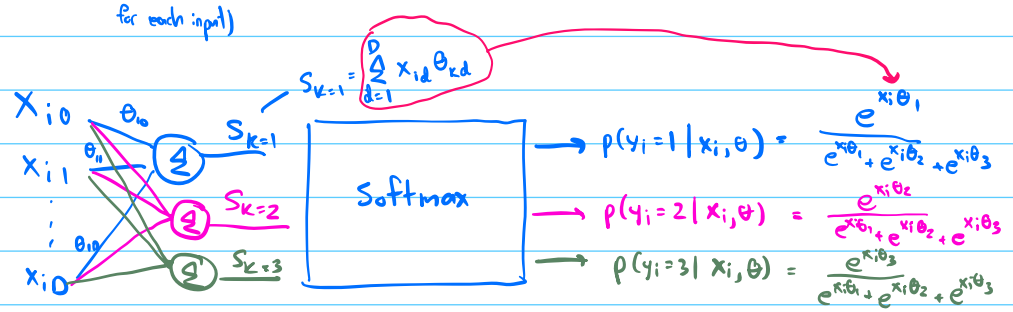


Softmax (use as activation function for Multiclass Classification Problem)

→ Output probability for each class

$y_i \in \{1, 2, \dots C\}$ C Classes





weight that is correspond to that output class (score)

$$p(y_i | x_i, \theta) = \frac{e^{x_i \theta_{y_i}}}{\sum_{k=1}^{C} e^{x_i \theta_k}}$$ sum of all the softmax output of class per 1 input row

For Numerical Stability = $\frac{e^{s_{y_i} - s_{max}}}{\sum_{k=1}^{C} e^{s_k - s_{max}}}$ to reduce computation in the case that each $e^{s_{y_i}}$ is very big

Cross Entropy Loss (Maximum Likelihood Estimate)

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_{k=1}^{C} e^{s_k}}\right)$$

normally have y_{ik} here but bc of one-hot bits encoding it's 1 for value of predict label = true label

Hinge Loss (SVM Loss)

$$L_i = \sum_{k=1, k\neq y_i}^{C} \max(0, s_k - s_{y_i} + 1) \rightarrow \text{stop learning when } L_i < 0$$

if there's several model with Loss < 0, then we cannot diff. which model is better (the score for the correct predicted label is clearly higher) cuz all the loss is set to 0.

CE can show better which model is better (closer to 0), CE always to improve

Hinge Loss saturates (no longer optimize at some point)

Regularization

- Change with respect to weight θ

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + \lambda R^2(W)$$

If use L-2 Reg = $\sum w_i^2$

L1-Reg $- \lambda R(W) = \sum_{i=1}^{D} |w_i|$

L2-Reg $- \lambda R^2(W) = \sum_{i=1}^{D} w_i^2$

Activation Function

On linear layer (score $s = xW + b$), X and W is matrix, the deriv are

dout = upstream derivative

$$\frac{\partial L}{\partial x} = dout \times W^T \qquad \frac{\partial L}{\partial w} = X^T \times dout \qquad \frac{\partial L}{\partial b} = dout$$

when deriv matrix-matrix mult → get transpose, vector-vector mult → normal or matrix-vector

Sigmoid → $y' = \frac{1}{1+e^{-s}}$ $\frac{\partial y'}{\partial w} = y'(1-y')$

Disadvantage → If s is very high or low, the gradient is almost 0 (saturate neuron cause vanishing gradient)

→ Output is always positive (not zero-center)

→ all of the weight update will either be positive or negative $\left(\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \sigma}\cdot\frac{\partial \sigma}{\partial s}\cdot\frac{\partial s}{\partial w} = \hat{y}(\hat{y}-y)\cdot x\right)$

so it does not depend on the weight, so all weight will always be increase or decrease based on $\hat{y}(\hat{y}-y) > 0$ or < 0

→ Create a zig zag path toward optimal w

TanH → Advantage → zero center

Disadvantage → Still saturate (gradient move slow when score are very high or low)

ReLU → $\sigma(x) = \max(0, x)$

→ Advantage → Fast convergence

→ Does not saturate (large and consistant gradient)

→ Disadvantage → If score <0, then ReLU =0 then you cannot improve and get out (Dead ReLU)

→ give some slightly positive biases (0.01) to make ReLU stay active for most inputs

↳ called Leaky ReLU if $\alpha = 0.01$ always

Parametric ReLU → $\sigma(x) = \max(\alpha x, x)$

↳ not result in dead ReLU

α = hyperparameter that can be train via backpropagation to have diff number

Advantage → Gradient do not die

Maxout Units → $= \max(w_1^T x + b_1, w_2^T x + b_2)$

Advantage → can create any approximate of an activation function with enough combination of Maxout Unit

→ Generalization of ReLU, Linear, do not die, not saturate

Disadvantage → Each Maxout Unit require new set of weight → Increase # of parameters

Weight Initialization → good initialise can result in better local minimum

If all weight initial with 0/ same value → Get same score, same gradient → No symmetry breaking, all neuron behave the same

If initialise with Gaussian with Small Random Number ($\mu=0, \sigma^2=0.01$)

↳ small weight cause small output → output approach 0 very fast for each layer in forward pass

↳ b/c the input for the next layer is small, the gradient (which uses the input for calculation) will be very small as well

↳ or output of last layer

and approach 0 with the deeper layer. ⇒ Vanishing gradient cause by small output

If initialise w/ Gaussian with Big Number ($\mu=0, \sigma=1$)

↳ b/c the weight is large, the score is large aswell

↳ if we use tanH, the large score will cause saturate neuron and the gradient will move very little (close to 0) → Vanishing gradient caused by saturated activation function (for tanH)

Xavier Initialisation ($\mu=0, \sigma^2=\frac{1}{n}$)

calculate for each layer → $Var(s) = Var\left(\sum_{i=1}^{n} x_i w_i\right) = \sum_{i=1}^{n} Var(x_i w_i) = n(Var(x)\,Var(w))$

(n = # of input (based on each layer))

Goal: we want $Var(s) = Var(x)$ → $Var(x) = n(Var(x)\,Var(w))$

(variance of output = input)

$\frac{1}{n} = Var(w)$ n = # of input neuron for layer l

So Xavier w/ tanh will make the output stay Gaussian distribution

But Xavier do not work with ReLU cuz ReLU kill half the data

For ReLU, we have to use Kaiming Initialisation = $Var(w) = \frac{2}{n}$ (ReLU kill half the data, so we use half the input, $\frac{n}{2} \rightarrow \frac{1}{\frac{n}{2}} = \frac{2}{n}$)

Exercise: Autoencoder

- unsupervised learning
- NN that is train to copy its input to its output
  - consist of encoder and decoder
    - encoder learn feature
  - layer between input and output has smaller size (latent space)

Lecture 8 - Training Neural Network Part 3

Data Augmentation → One way to better generalise, type of regularization

You always crop the image first, to have image dimension as input of what you want

1. Flipping
2. Crop + Flip → Do multiple crop on diff region of the image (crop to the dim of input you want) and then flip all of them.
   → More robust to partial exclusion
3. Random Brightness and Contrast change
4. Random Crop → Random crop on training
   → Fixed crop on testing

Use same data from data augmentation when compare NN

Advanced Regularization → Another way to improve generalise

1. Add weight decay $\Rightarrow \theta_{k+1} = \theta_k - \alpha \nabla_\theta L - \lambda \theta_k$ → so $\theta_{k+1} = \theta_k - \alpha \left(\nabla L(\theta_k) + \nabla \lambda R(\theta_k) \right)$
   → If this is L-2 Reg → gradient of regularization $= \frac{\partial}{\partial w} \sum_{i=1} w_i^2$
2. Early Stopping ⇒ Stop after validation loss increase n times in a row.

3. Ensemble ⇒ Train multiple model in diff way (diff activation, augmentation, loss, etc.) that lead to diff local optima and average it.
   Expected combined error should decrease linearly w/ ensemble size ⇒ provided errors are uncorrelated

2 type of ensemble

→ 1. Bagging ⇒ build diff model like ensemble → divide training data into k parts (can be overlap)

2. Dropout ⇒ disable a random set of neurons (inputs) (do it during the forward pass) (random new set for each minibatch)
   → reduce capacity of NN

Consider as ensemble technique bc training large ensemble of models, each new model w/ diff minibatches w/ shared parameter

bagging melted w/ shared param.

Advantage → prevent one node from being too specialise, make other node learn other feature as well => More Robust

Disadvantage → another hyperparameter (% of dropout) that need to be optimize
   → Need larger model + more training time

On test time ⇒ turn on all neuron → no dropout for validation & test data

→ Need to do weight scaling inference rule during training
   → Adapt weight to match the condition at training time (multiply weight with dropout prob.)

$$\mathbb{E}(s) = p\left(\sum \theta_i x_i\right)$$

↑ $i=1$

keeping probability (or $1-p$ if p = dropping probability)

Batch Normalization

Want to do Normalization after finishing the layer to not have saturate neuron

1. Normalise mean and variance of the input to the activation function w/ the mean & var of minibatch

$$\hat{x}^k = \frac{x^k - E(x^k)}{\sqrt{Var(x^k)}}$$

$E(x^k)$ ← mean of minibatch over feature k

$Var(x^k)$ ← variance of

Batch Normalization layer → apply after Fully Connected Layer (FC) or CNN and before non-linear activation function

2. Allow Network to change the range of x^k during backprop

$$y^k = \gamma^k \hat{x}^k + \beta^k$$

γ^k, β^k ← parameter that could learn to undo normalization $\hat{x}^k \rightarrow x^k$

if $\gamma^k = \sqrt{Var(x^k)}$ $\beta^k = E(x^k)$

Advantage → Converge to optima faster
→ got much deeper network → more stable gradient
→ Reduce internal covariate shift → Change in input distribution

At test time → Compute μ and σ^2 by running an exponentially weighted averaged across training minibatches

$Var_{test} \rightarrow Var_{running} = \beta Var_{running} + (1-\beta) Var_{minibatch}$

$\mu_{test} \rightarrow \mu_{running} = \beta \mu_{running} + (1-\beta) \mu_{minibatch}$

} use this μ and σ^2 to normalise test data instead

BN get worse performance when batch size is smaller

Other Normalisation → Layer Norm, Instance Norm, Group Norm

Lecture 9 : CNN

FC use too much hyperparameter from weight, so we need to
Restrict degree of freedom (apply some inductive bias → look for specific feature/ thing)
↳ Weight sharing, extract same features independent of location

Convolutions

Filter Kernel → a matrix that's slide across input matrix
→ Equal to dot product btw filter weight w and input x_i-th chunk (output for each chunk = 1x1)

z_i (output) $= w^T x_i + b$

→ Need to have same depth as input (if input is RGB, then filter need to have depth = 3, result depth = 1)
dimension
→ Output dim is depend on
1. Size of kernel (the bigger, the smaller output dim)
2. # of stride (reduce output dim)
3. padding

↳ multiply weight in each depth to each input in chunk and sum it all up

$(x_1 w_{11} + x_2 w_{12} \dots) + (x_1 w_{21} + x_2 w_{22} \dots) + (x_1 w_{31} + \dots x_2 w_{32} \dots)$

* The weight inside filter kernel is a hyperparameter that need to be learn
so kernel w/ 3 depth and size 5x5 has # of weight/params = 3x5x5 = 75

Convolution Layer

→ Apply different filter with diff. weights
→ Resulting layer from all the filter kernel to input = Activation maps (DxWxH of output)
→ Number of depth in Activation maps = Number of filter kernel apply to input (w/ same dim)
→ Filters that apply to input form a Convolution layer (n filter x WxH of filter)
together
(depth is implicitly given by input depth)

Stride

→ place filter S pixel from previous input
→ Can have illegal stride (filter does not fit, there's a remainder)

Output dim $= \left(\frac{N-F}{S}+1\right) \times \left(\frac{N-F}{S}+1\right)$ S = Stride
N = H or W dim of input F = Hor W of filter

If there's a fraction, then the stride is illegal → Can be solved by padding

(mb the other one)

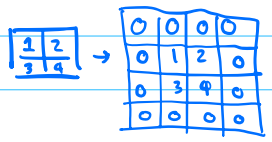
Shrinking dim too fast is not a good idea → so we apply padding to not reduce dim

Padding

→ Use to stop dimension from shrinking too quickly

→ Also allow corner pixel to be use in calculation more than once

→ Most common = zero padding (add 0 pixel all around allow the chunk)



$$\text{Output Size} = \left(\frac{N+2P-F}{S}+1\right) \times \left(\frac{N+2P-F}{S}+1\right)$$

if fraction, then do floor operation

Type of Convolution → 1. Valid Convolution = no padding

2. Same Convolution = input dim = output dim → Set padding $P = \frac{F-1}{2}$

Example - Input image = 32×32×3

10 filter 5×5

Stride = 1

Padding = 2

$$\text{Output} = \frac{N+2P-F}{S}+1 = \frac{32+2(2)-5}{1}+1 = 31+1 = 32$$

Output = 32×32×10

No. of params per filter = W×H×D + 1 bias = 5×5×3+1 = 76 (each filter has 1 bias)

Total params for this Convolution layer = # of filter × params per filter = 10×76 = 760 params

Convolution Neural Network

Input → Convolution layer → Activation function → Convolution layer (use output of Activation function as input → Activation → .... → FC Layer (in the end)

(do Convolution with n filters, then apply activation function to it) w/ dim depth of # of filter on last Convolution layer)

Pooling Layer → reduce size of input (downsampling), no learning (no weight)

→ Come after the Convolution layer (Conv layer = Feature Extraction, Pooling layer = Feature Selection)

→ **Max pooling** → A filter that only pick max value of the chuck as the output (select the strongest feature in the region)

→ **Average pooling** → A filter that average value from the chunk

Pooling layer has 2 hyperparams

1. Filter dim F (F×F)

2. Stride S

Output of Pooling layer

$W_{out} = \frac{W_{in} - F}{s} + 1 \qquad H_{out} = \frac{H_{in} - F}{s} + 1 \qquad D_{out} = D_{in}$

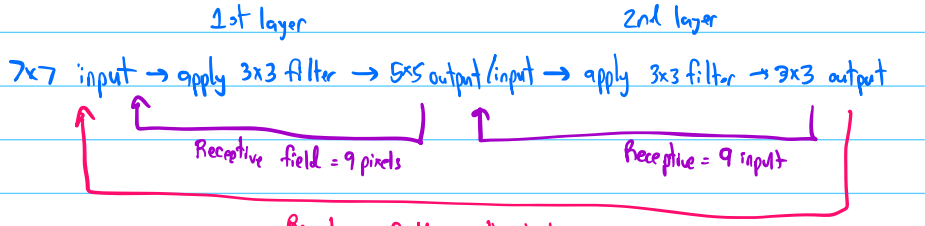
The Final FC Layer → Make decision w/ the extracted feature from convolutions

Receptive Field (Spatial extend of the connectivity of a convolutional filter)

How many input space were used to get this output pixel

If filter 3x3 apply on input, result in 1 pixel. Then then receptive field = 9 pixels

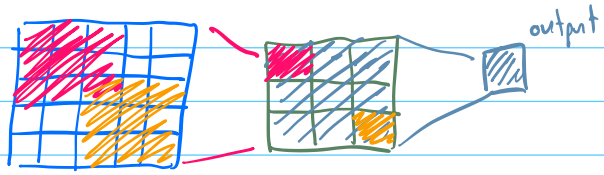
When we chain Convolution filter



1 pixel of output use 3x3 of 2nd layer

and to get 3x3 of 2nd layer required 5x5 (stride = 1) of 1st layer

∴ 5x5 = 25



* Note, it's more efficient to use many kernel than 1 big filter kernel (have less weight + More nonlinearity)

Spatial Batch Normalization

When using BN on Convolution layer/NN

input = N×C×H×W

We compute μ and var for each channel for all data in minibatch. (normally we normalise for each feature) ← Fully Connected

N×C×H×W → 1×C×1×1

If there's C channel, each with H×W dim, and there's N minibatch

then we get μ and var for each C using data for H×W×N (entire weight in the channel × all the data in minibatch)

C₁ use data from entire weight of channel 1 of all the data in minibatch

N×C×H×W → N×H×N×C → (N×H×W)×C

find μ and σ^2

Lecture 10 - CNN Part 2

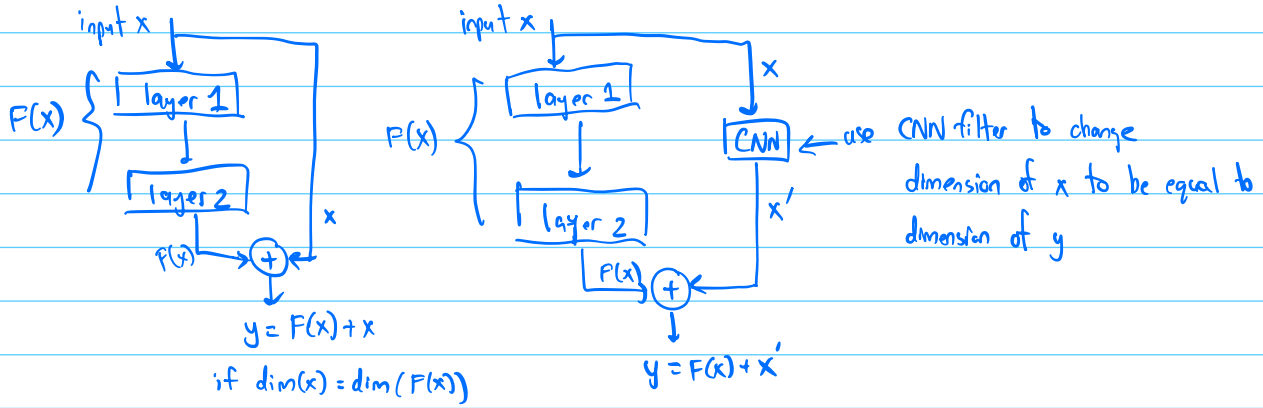
Classic Architecture

- **LeNet**
  - Digit Recognition (10 classes)
  - use sigmoid/tanh } not common anymore
  - use average pool
  - As NN is deeper, W×H ↓, no. of filter ↑
- **Alex Net** (first architecture to use CNN with imagenet)
  - use large size filter → not common
    - As NN is deeper, W×H ↓, no. of filter ↑
  - use max pool (3×3, s=2)
  - use ReLU
  - 1000 classes
- **VGG Net**
  - more simplicity
  - use 3×3 Convolution filter, stride = 1 that result same dim for output (add padding) (always)
  - Max pool (2×2, s=2)
  - As NN is deeper, W×H ↓, no. of filter ↑

As we add more layer, training is harder + vanishing gradient + degradation

Skip Connection ← solution

Residual Block (Res Net) → With this technique, you can add alot more layer



So when there's a vanishing gradient (either from input or backprop) in the F(x) (F(x) close to 0), then we can still use value of x for further layer and ignore ("skip") output F(x)
(look like we're skipping the output)

$$x^{L+1} = f(y) = f(F(x) + x) = f(x) \text{ instead of } f(0) \text{ w/ no residual}$$

if F(x) = 0

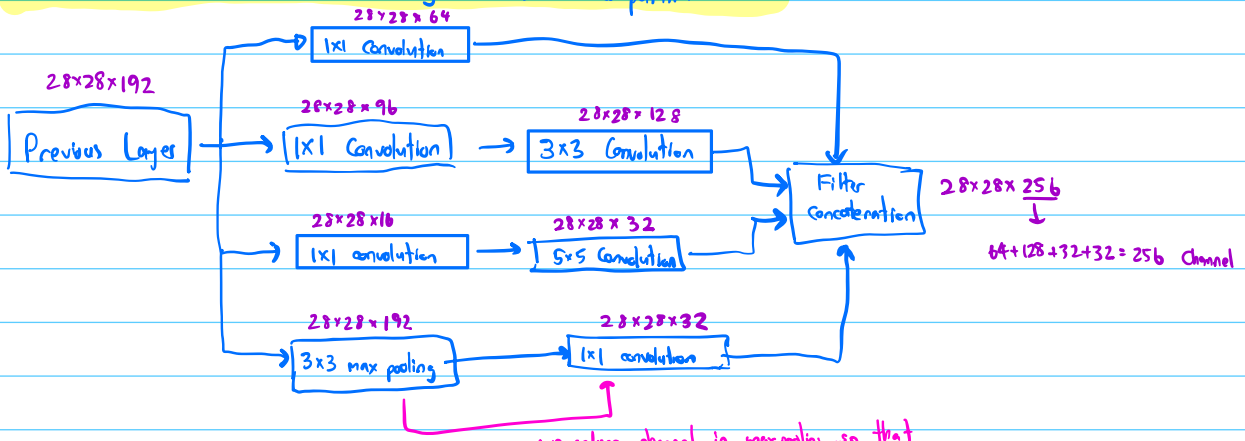
- Identity (input x) is easy for the residual block to learn ($x^{L+1} = x$)
- Not hurt the performance, can only improve

1x1 Convolutions

- Reduce or Increase the channel
- Using n filter of 1x1 Convolution is same as using n FC layer
  - Add non-linearity (not by itself but w/ activation func afterward)
  - When use in last layer instead of FC to get output → # of output = # of filter

Inception Layer (GoogleNet)

- Learn best weight when training by using multiple convolutions (kernel) size with multiple filters and pooling layers in parallel and concat all layer together. This help NN find more useful feature
- Use 1x1 convolution to reduce dimensionality (channel) and computational cost



Xception Net (extreme version of Inception)

- Use depthwise Separable Convolutions
- 36 Conv layer w/ skip connection
- Depthwise Separable Convolution = Filter apply only on certain depth

  If input has D channel, then use D filter of WxHx1 (1 channel) => DxWxHx1

  Output has same # of channel as input

  Then use N filter of 1x1xD to turn into N Channel → Nx1x1xD

  * normal conv would use N filter of WxHxD → NxWxHxD

  (help reduce computational cost)

Fully Convolutional Network

- We can convert Fully Connected layer into Convolutional Layer
- 1 × 1 × D Convolution w/ N filter = Fully Connected layer w/ N output (# of filter = # of output)
- When use FC Layer → size of input is fixed (take same # of inputs)
- When use Convolution layer → 1×1 does not care about size, so input is flexible

FCN (Convolutional) : Semantic Segmentation

- same output size as input (but # of channel of output = # of class want to predict)
- go from some little input size (from convolution) to big output size (Upsampling)
- Give score of classes for each pixel → "void class" for unlabelled pixels

Type of Upsampling

First, double the size of input w/ blank space in between each input, then ...

1. Interpolation - Increase the size using
   1. Nearest Neighbor (look at value of neighbor)
   2. Bilinear (weighted average of pixel)
   3. Bicubic

2. Transposed conv - Increase size w/ convolution - unpooling
   - Convolution filter (learned) (up-convolution)
     ↑ add value to blank space using filter

U-Net (Autoencoder using Convolution, pooling, up-convolution, and skip connection)

Use skip connection to add details back to decoder so that it has more detail to work with (spatial dim)

Encoder (Contraction Path)
- Height & Width ↓, Depth ↑
- Use Conv & max pooling to down sampling
- ReLU

Decoder (Expansion Path)
- Height & Width ↑, Depth ↓
- 2×2 Up conv → double input
- Skip Connection → Concatenate feature maps of encoder to decoder → then 2 3×3 Convs → final layer 1×1 Conv to map channel to # classes

* input and output w x H has to be the same

Lecture 11 - Recurrent Neural Network and Transfer

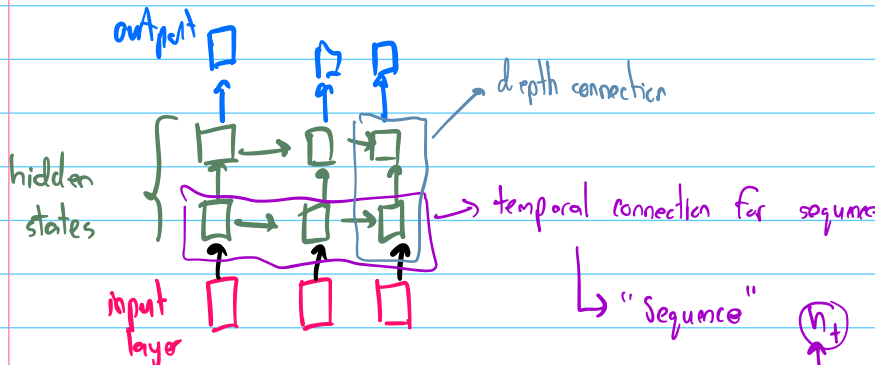
Transfer Learning

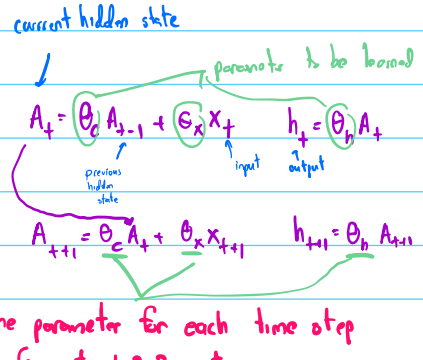
- We don't have enough resource or data → use Pretrain model instead
- Pretrain should have similar low level feature or same input
- Use the pretrain feature extraction (CNN layers + FC layer)
- Replace last FC layer to have correct output classes (If have large dataset, we can train more FC layer w/ low learning rate)

Recurrent Neural Network

- Input / Output can be sequence (set of inputs = dependent to eachother)
- Flexible
  - One to one (classic NN for image classification)
  - One to many (image captioning - one image input, output as a sentence)
  - many to one (language recognition - what language is this sentence)
  - many to many (Machine translation - translate language to other language)
    → start produce output after all the input are processed (require all info first)
  - many to many (Video classification)
    → can produce first output right away with first input

Multi layer RNN





RNN has long term dependency (A_t is depend on A_0) and since weight is always the same, it became exponentially big

$$A_t = (\theta_c)^t A_0 \quad \text{b/c } A_1 = \theta_c A_0 \quad A_2 = \theta_c A_1 = \theta_c^2 A_0 \quad A_3 = \theta_c A_2 = \theta_c^3 A_0 \dots$$

Problem of RNN

if θ_c is very small (<1) $\rightarrow \theta^t \rightarrow 0$ $\therefore$ vanishing gradient (also can vanishing from activation function sigmoid or tanh)

if θ_c is large (>1) $\rightarrow \theta^t \rightarrow \infty$ $\therefore$ exploding gradient $\rightarrow$ fix by using gradient clipping (if value $> x$, use x)

θ_c is matrix $\rightarrow$ eigen decomposition $\theta_c^t = Q \Lambda^t Q^T$

Make eigenvalue = 1 (weight) $\rightarrow$ Allow Cell to maintain state $\rightarrow$ Long Short Term Memory (LSTM)

Long Short Term Memory (LSTM)

Setting eigenvalue = 1 to avoid vanishing gradient problem

- use tanh
- Cell - transport info through the Unit
- Gate - remove or add info to the cell state

3 Type of Gate

1) Forget Gate $\rightarrow$ Use sigmoid = output between 0 (forget) and 1 (keep)
   - use x_t (input of current) and h_{t-1} (past hidden state)
   - multiply output to cell state (output = 0.7 $\rightarrow$ keep 70% of info)

$$f_t = sig(\theta_{xf} x_t + \theta_{hf} h_{t-1} + b_f)$$

2) Input Gate $\rightarrow$ Decide which value will be update
   - Use sigmoid to check if output (tanh) will be add to cell state or not

$$i_t \text{ (input)} = sig(x_t \theta_{xi} + h_{t-1} \theta_{hi} + b_i)$$

- output (g_t) tanh will be between $(-1, 1)$

$$C_t = f_t \odot C_{t-1} + i_t \odot g_t$$

(C_{t-1}: previous cell state; f_t: forget gate; i_t: input gate (forget or not); g_t: output from tanh $\rightarrow g_t = \tanh(x_t \theta_{xg} + h_{t-1} \theta_{hg} + b_g)$)

3) Output Gate $\rightarrow$ Decide which values will be output (and use in next cell as a hidden state)

$\rightarrow O_t = sig(x_t \theta_{xo} + h_{t-1} \theta_{ho} + b_o)$ $\rightarrow$ decide if we will send hidden state h_t to the next cell

$$h_t = O_t \odot \tanh(C_t)$$

$\rightarrow$ Produce Actual Output and pass to next time step

All weight and bias are the same for each time step and are learned through backprop

How LSTM solve vanishing gradient

$$C_t = f_t \odot C_{t-1} + i_t \odot g_t$$

1. Vanishing from weight (f_t)
   f_t use sigmoid = 1 for important info, so when you get important info in the past, there will be no exploding / vanishing gradient from exponential

2. Vanishing from activation function (tanh = g_t)

   → if g_t has vanishing gradient, we still has $f_t \odot C_{t-1}$ to propagate backward further (similar to residual connection)

Knowledge from past exam

21ss

1. Maximum Likelihood Estimate → supervised learning
2. ↑ Batch sie ↑ Learning rate , ↓ Batch size ↓ LR (more confidence)
3. Batch Norm → formula add non linearity
4. non suitabe for activation function → change value of input to function
5. ~~Batch Norm is not regularization~~ it's regularization
6. Convolution — translation equivariant ($f(g(x)) = g(f(x))$)
   - Not rotation equivariant → need data augmentation to learn picture of rotation
   - Convolution layer is not translation invariant but
     Other part of CNN might cause output to be translation invariant such as pooling layer

BCE & CE not use $\frac{1}{N}$

only $\hat{\sum_{i=1}} \overset{K}{\sum_{j=1}} y_{ij} \log \hat{y}_{ij}$

or $\hat{\sum_{i=1}} y_i \log \hat{y}_i + (1-y_i) \log (1-\hat{y}_i)$