The Last Few Bits..

CS771: Introduction to Machine Learning

The Attention Mechanism

$$\alpha_{nm}(\boldsymbol{X}) = \frac{\exp(\boldsymbol{q}_n^\top \boldsymbol{k}_m)}{\sum_{j=1}^{N} \exp(\boldsymbol{q}_n^\top \boldsymbol{k}_j)}$$

$$\boldsymbol{h}_n = \sum_{m=1}^{N} \alpha_{nm}(\boldsymbol{X}) \boldsymbol{v}_m$$

$\boldsymbol{Q}$ and $\boldsymbol{K}$ are assumed $N \times d$

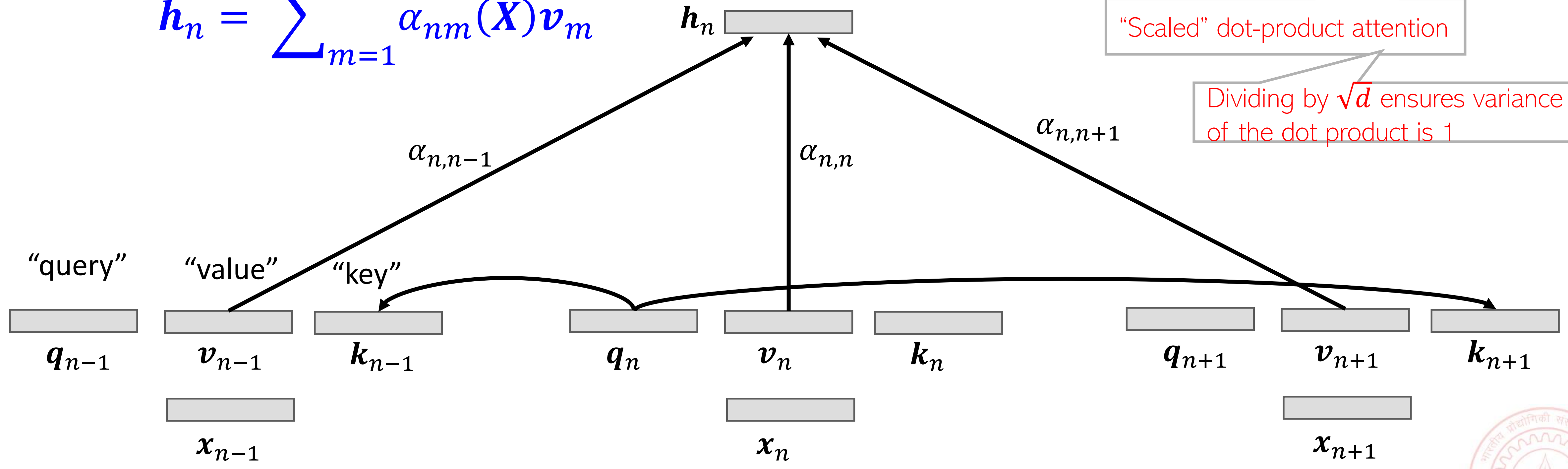
$N \times v$

$N \times v$

$$\boldsymbol{H} = \operatorname{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d}}\right)\boldsymbol{V}$$

"Scaled" dot-product attention

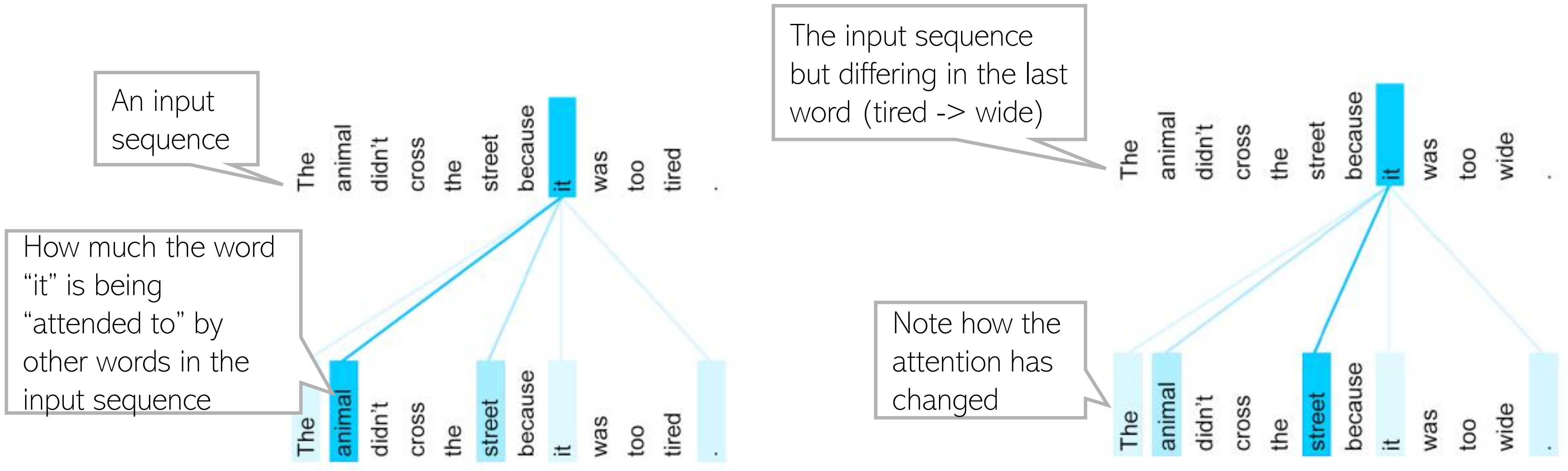
Dividing by $\sqrt{\boldsymbol{d}}$ ensures variance of the dot product is 1



$$\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}_Q \qquad \boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}_K \qquad \boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}_V$$

(Self-)Attention: An Illustration

- With self-attention, each token $\boldsymbol{x}_n$ can “attend to” all other tokens of the same sequence when computing this token’s embedding $\boldsymbol{h}_n$


An input sequence
The input sequence but differing in the last word (tired -> wide)
How much the word “it” is being “attended to” by other words in the input sequence
Note how the attention has changed
The animal didn't cross the street because it was too tired .
The animal didn't cross the street because it was too wide .

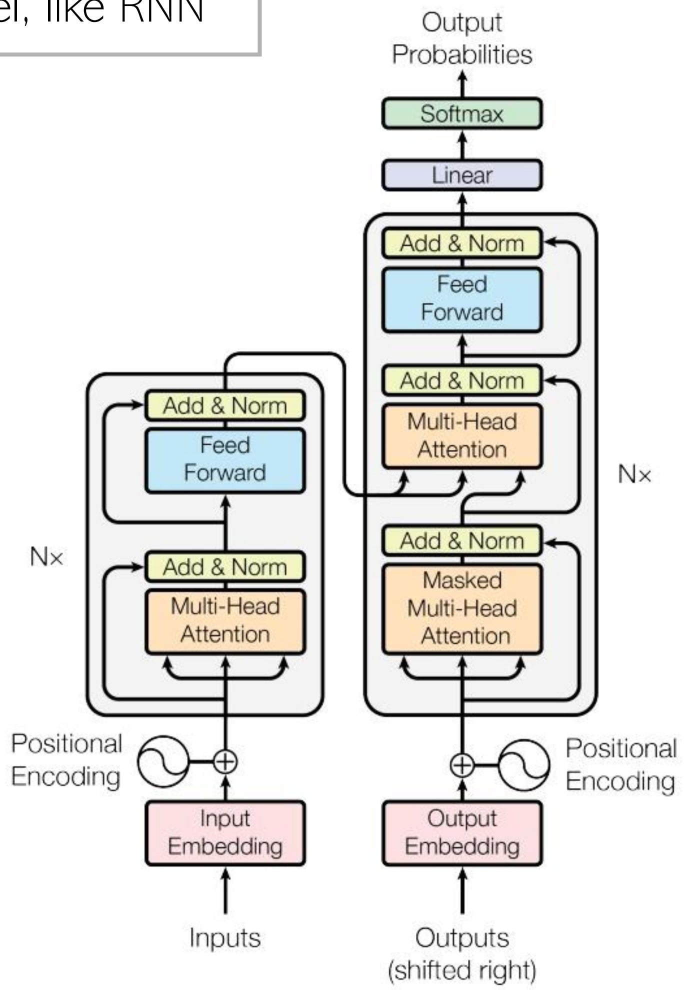
- Attention helps capture the context better and in a much more “global” manner
  - “Global”: Long ranges captures and in both directions (previous and ahead)

Example credit: https://blog.research.google/2017/08/transformer-novel-neural-network.html

Transformer

Basically a sequence-to-sequence model, like RNN

- Maps an input token sequence to an output token sequence
- Uses the idea of attention
  - "self-attention" over all input tokens
  - "self-attention" over each output token and previous tokens
  - "cross-attention" between output tokens and input tokens
- Transformers also compute embeddings of all tokens in parallel
- Transformers are based on the following key ideas*
  - "Self-attention" and "cross-attention" for computing the hidden states
  - Positional encoding
  - Residual connections
- Attention helps capture the context better and in a much more "global" manner in sequence data



"Attention is all you need" (Vaswani et al, 2017)

Positional Encoding

- Transformers also need a "positional encoding" for each token of the input since they don't process the tokens sequentially (unlike RNNs)
- Let $\boldsymbol{p}_i \in \mathbb{R}^d$ be the positional encoding for location $\boldsymbol{i}$. One way to define it is

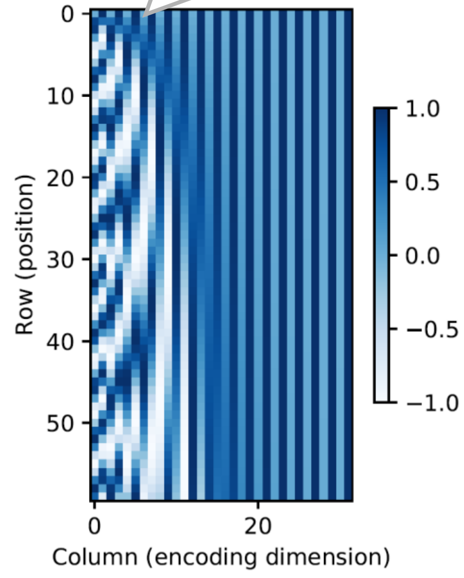
Here $\boldsymbol{C}$ denotes the maximum possible length of a sequence

$$p_{i,2j} = \sin\left(\frac{i}{C^{2j/d}}\right), \quad p_{i,2j+1} = \cos\left(\frac{i}{C^{2j/d}}\right)$$

Positional encoding vector for location $\boldsymbol{i}$ assuming $\boldsymbol{d} = 4$

$$\boldsymbol{p}_i = [\sin(\frac{i}{C^{0/4}}), \cos(\frac{i}{C^{0/4}}), \sin(\frac{i}{C^{2/4}}), \cos(\frac{i}{C^{2/4}})]$$

Note the smooth transition as the position index changes



- Given the positional encoding, we add them to the token embedding

$$\hat{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{p}_i$$

- The above positional encoding is pre-defined but can also be learned

Pic credit: PML-1 (Murphy, 2023)

Zooming into the encoder and the decoder..

FF operation is applied "position-wise" (for each token separately) but all FF blocks in the layer have same weights

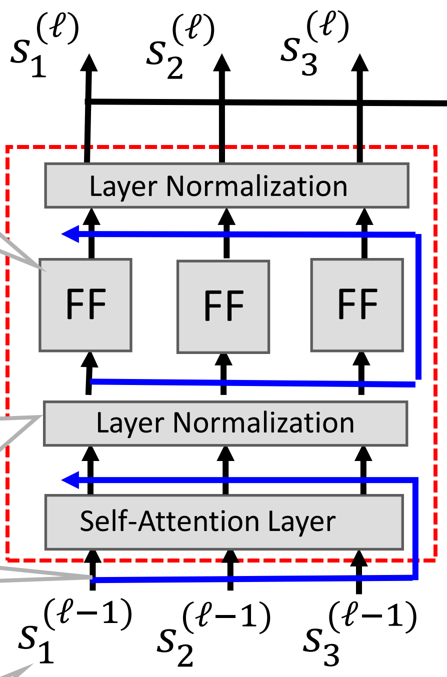
Each FF (feed-forward) is usually a linear layer + ReLU nonlinearity + another linear layer

Layer normalization (batch normalization is difficult since difference input sequences can be of different lengths)

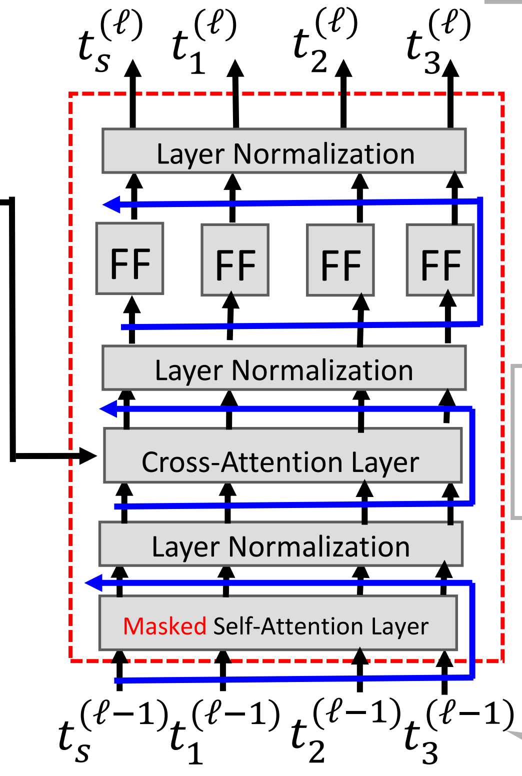
Blue arrows are the residual connections

Input ("source") token embeddings

Fixed in the first layer (obtained from an input embedding table) and learned for subsequent layers



An Encoder Block (N such blocks)



A Decoder Block connected with the corresponding encoder block (N such blocks)

Output ("target") token embeddings

t_s is a special start of sequence (SOS) token

Most likely output token at step m

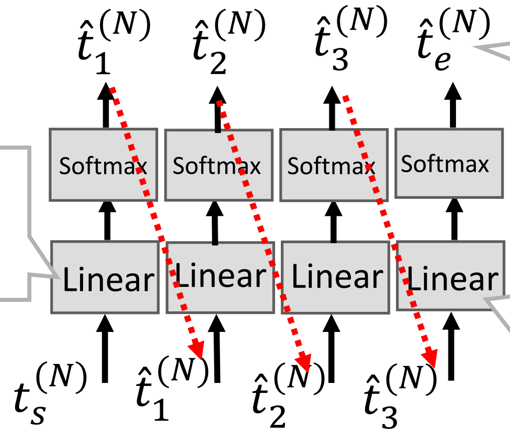
"auto-regressive" generation

Output $\hat{t}_{m-1}^{(N)}$ from the previous step $(m-1)$ in the sequence

$$\hat{t}_m^{(N)} = \text{argmax}_{i=1,\dots,V}\ \text{softmax}(\boldsymbol{W} t_m^{(N)})$$

t_e is a special end of sequence (EOS) token

Like FF, linear operation is also applied position-wise



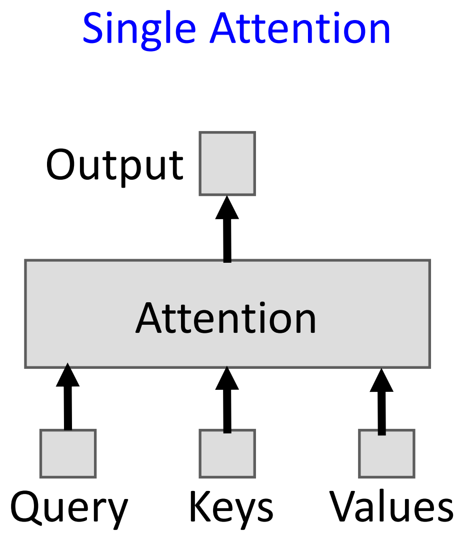
Decoder's output layer

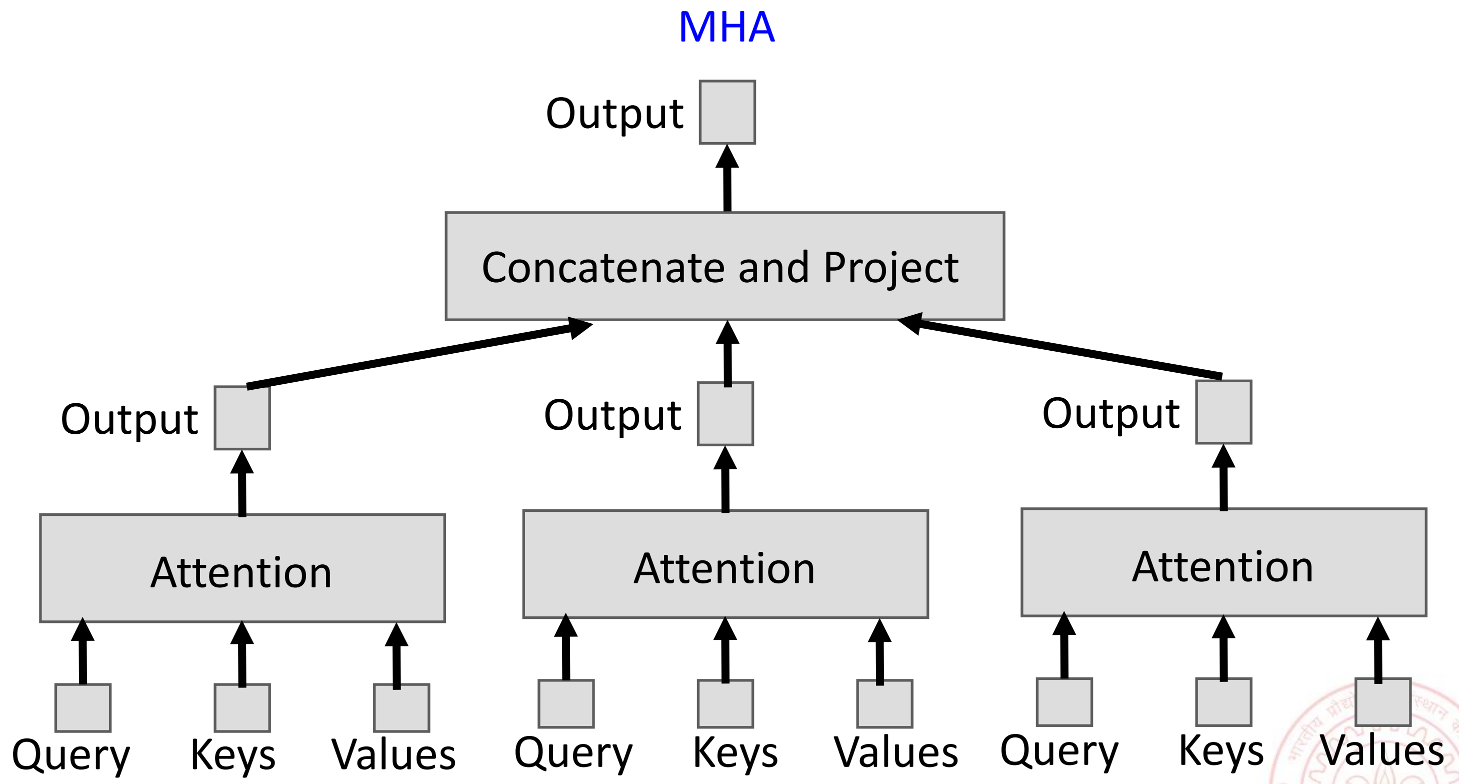
With weight matrix W of size $V \times D$ where V is vocab size and D is the dimensionality of the last decoder block embeddings $t_m^{(N)}$

Note the one position (towards right) shift between decoder's input vs output

Multi-head Attention (MHA)

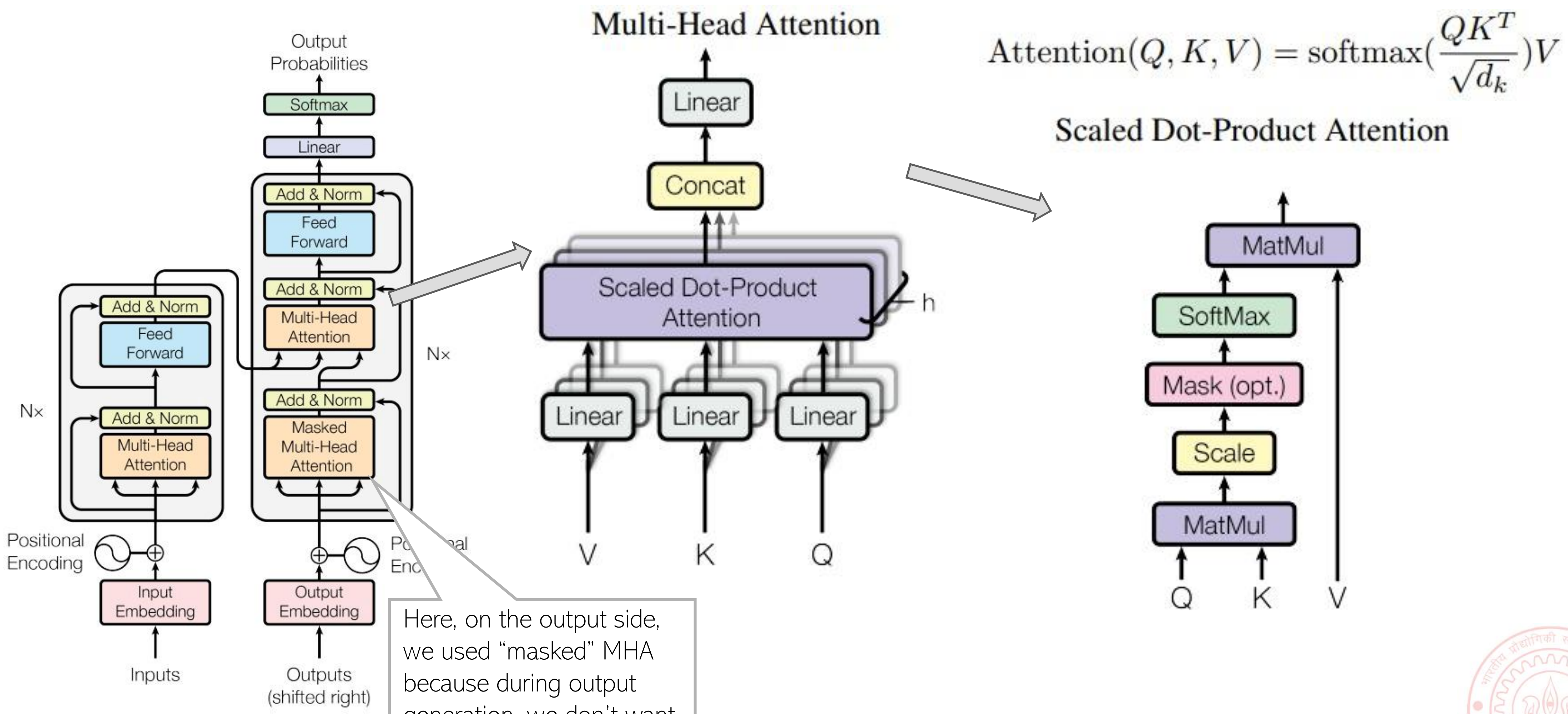
- A single attention function can capture only one notion of similarity
- Transformers therefore use multi-head attention (MHA)





Pic credit: https://huggingface.co/learn/nlp-course/chapter1/

(Masked) Multi-head Attention (MHA)



$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Here, on the output side, we used "masked" MHA because during output generation, we don't want to look at future tokens

Pic source: "Attention is all you need" (Vaswani et al, 2017)

Computing Attention Efficiently

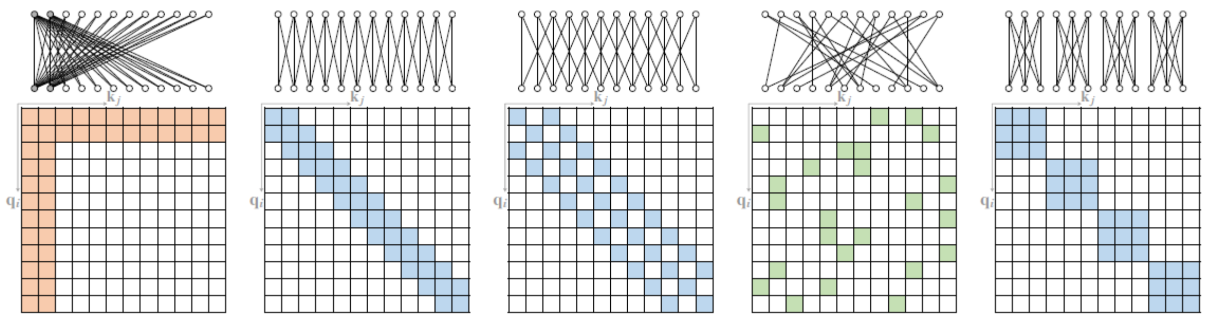
- The standard attention mechanism is inefficient for large sequences

$O(T^2)$ storage and computation cost for a T length sequence

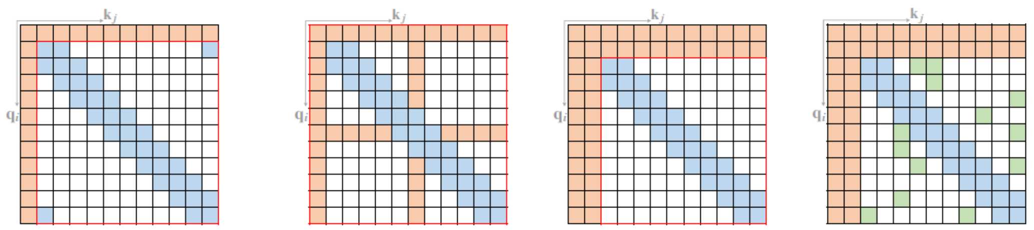
$$\boldsymbol{H} = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d}}\right)\boldsymbol{V}$$

- Many ways to make it more efficient, e.g.,

Sparse Attention



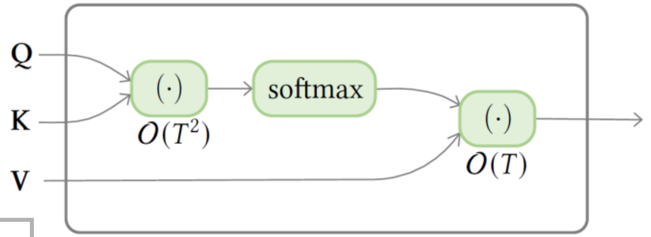
(a) global (b) band (c) dilated (d) random (e) block local



(a) Star-Transformer (b) Longformer (c) ETC (d) BigBird

Linearized Attention

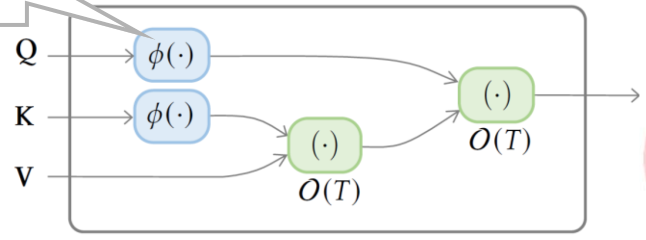
$$\exp(\boldsymbol{Q}\boldsymbol{K}^\top) \approx \phi(\boldsymbol{Q})^\top\phi(\boldsymbol{K})$$



(a) standard self-attention

A nonlinear projection based on kernels

E.g., kernel random features projection

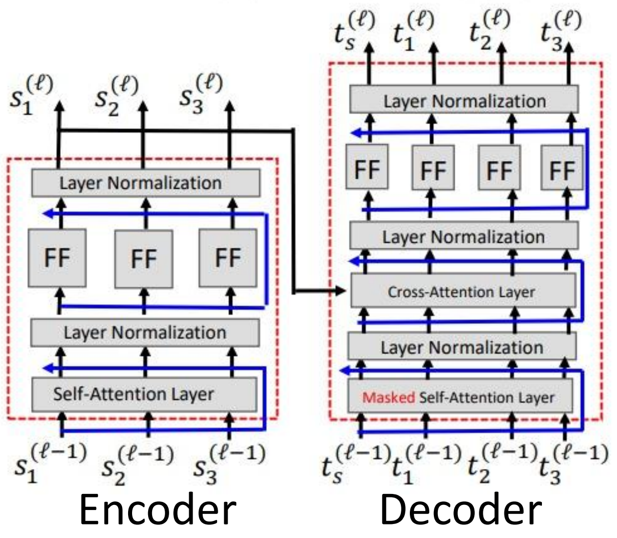


(b) linearized self-attention

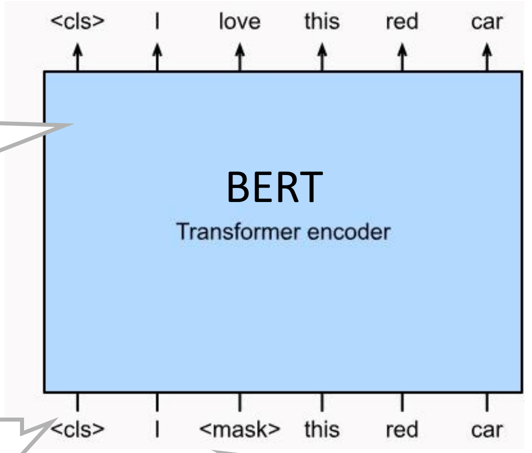
Pic source: A Survey of Transformers (Lin et al, 2021)

Popular Transformers Variants: BERT and GPT

- The standard transformer architecture is an encoder-decoder model
- Some models use just the encoder <u>or</u> the decoder of the transformer
- BERT (Bidirectional Encoder Representations from Transformers)
  - Basic BERT can be learned to encoder token sequences
- GPT (Generative Pretrained Transformer)
  - Basic GPT can be used to <u>generate</u> token sequences similar to its training data



$s_1^{(\ell)}$ $s_2^{(\ell)}$ $s_3^{(\ell)}$
Layer Normalization
FF FF FF
Layer Normalization
Self-Attention Layer
$s_1^{(\ell-1)}$ $s_2^{(\ell-1)}$ $s_3^{(\ell-1)}$
Encoder
$t_s^{(\ell)}$ $t_1^{(\ell)}$ $t_2^{(\ell)}$ $t_3^{(\ell)}$
Layer Normalization
FF FF FF FF
Layer Normalization
Cross-Attention Layer
Layer Normalization
Masked Self-Attention Layer
$t_s^{(\ell-1)}$ $t_1^{(\ell-1)}$ $t_2^{(\ell-1)}$ $t_3^{(\ell-1)}$
Decoder



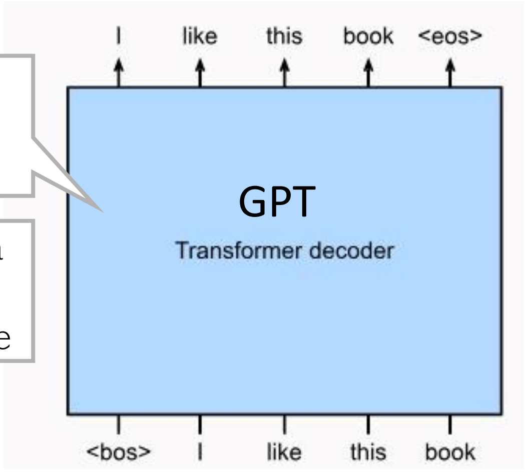
This encoder can be used for other tasks by fine-tuning

A transformer which contains only the encoder

Trained unsupervisedly using a missing token prediction objective

This is just start of sentence token

Missing token which BERT tries to predict



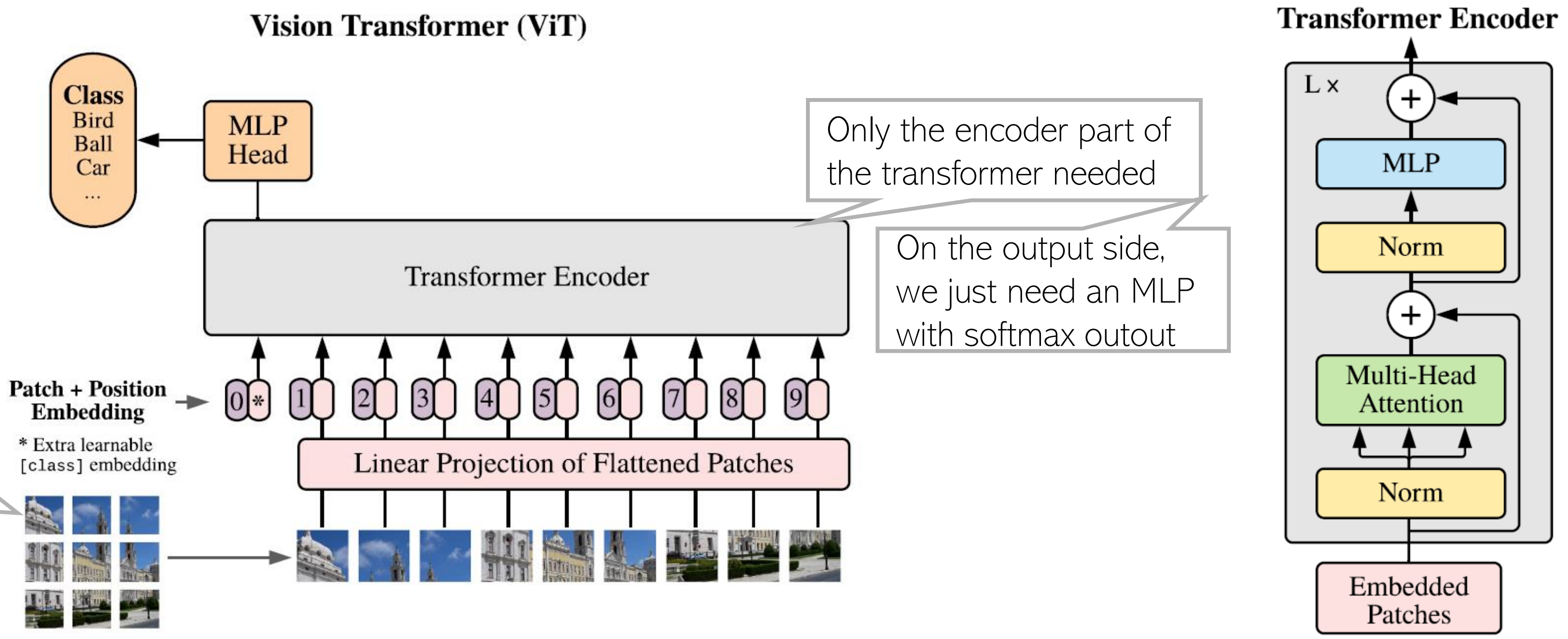
Also, no cross-attention since there is no encoder

A transformer which contains only the decoder

Pre-trained using a next token prediction objective

Transformers for Images: ViT

- Transformers can be used for images as well#. For image classification, it looks like this



Only the encoder part of the transformer needed

On the output side, we just need an MLP with softmax outout

Treat image patches as tokens of a sequence

Also use the position information

- Early work showed ViT can outperform CNNs given very large amount of training data
- However, recent work* has shown that good old CNNs still rule! ViT and CNN perform comparably at scale, i.e., when both given large amount of compute and training data

# An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale (Dosovitskiy et al, 2020)

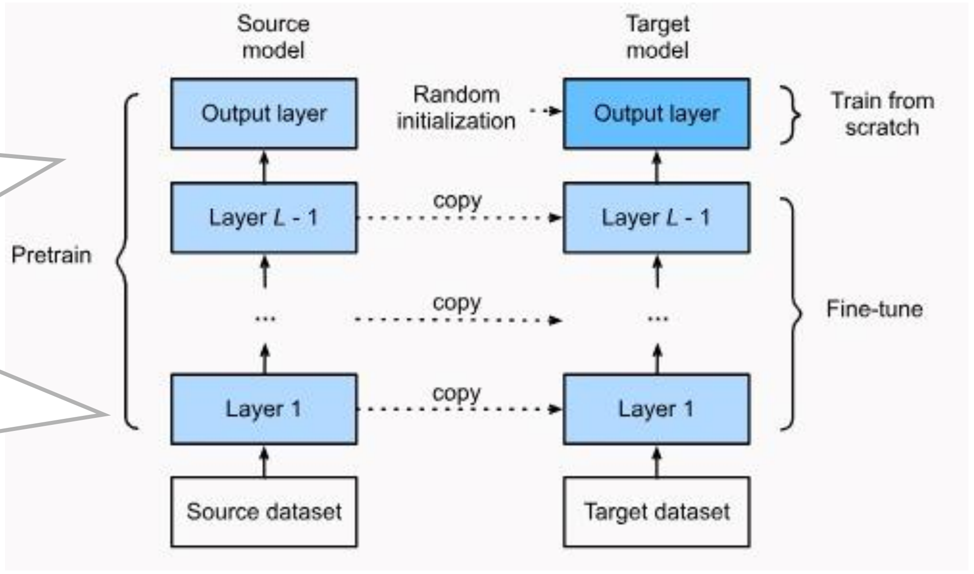
*ConvNets Match Vision Transformers at Scale (Smith et al, 2023)

Fine-tuning and Transfer Learning

- Deep neural networks trained on one dataset can be reused for another dataset
  - It is like transferring the knowledge of one learning task to another learning task
- This is typically done by "freezing" most of the lower layers and finetuning the output layer (or the top few layers) – this is known as "fine-tuning"

Initial model with frozen layers is called the "pre-trained" model and the updated model is called the "fine-tuned" model

This example is for an MLP like architecture but fine-tuning can be done for other architectures as well, such as RNN, CNN, transformers, etc



BERT (pre-trained in unsupervised manner) fine-tuned for a sentence classification task by adding a fully connect MLP to predict class-label of a sentence

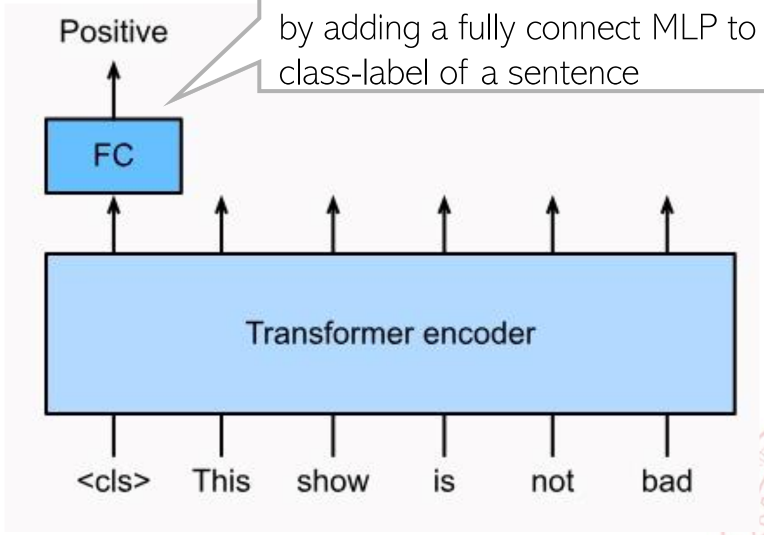
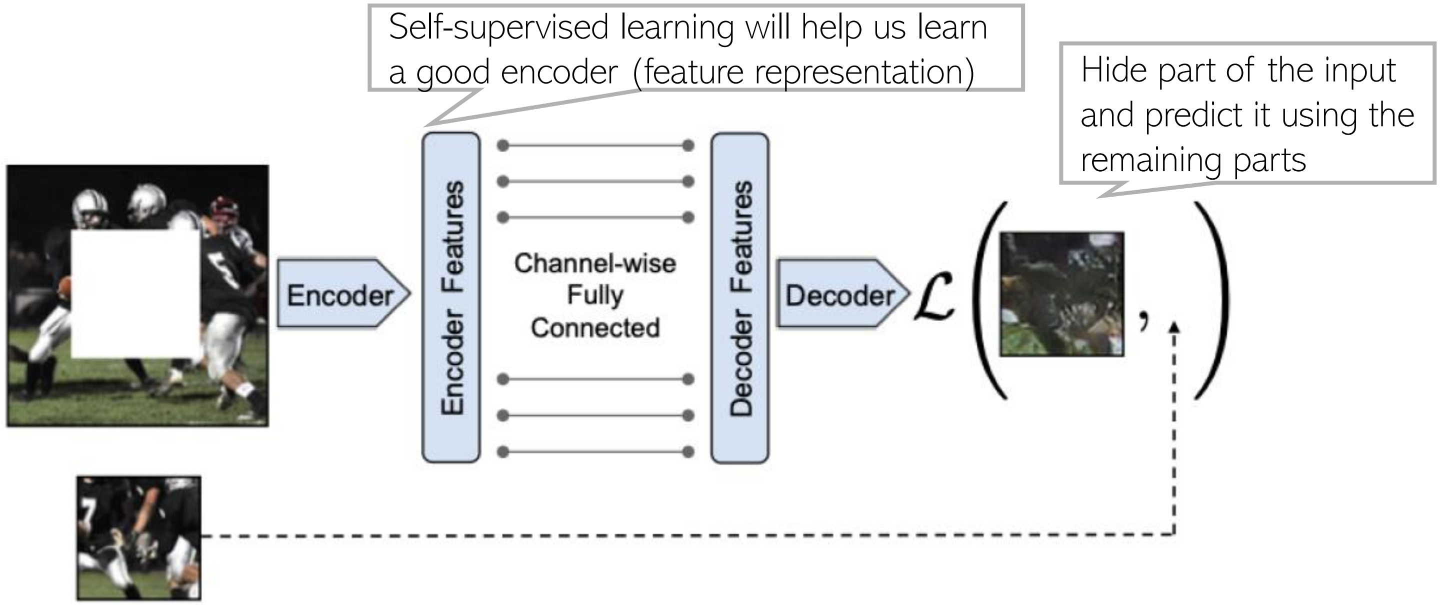
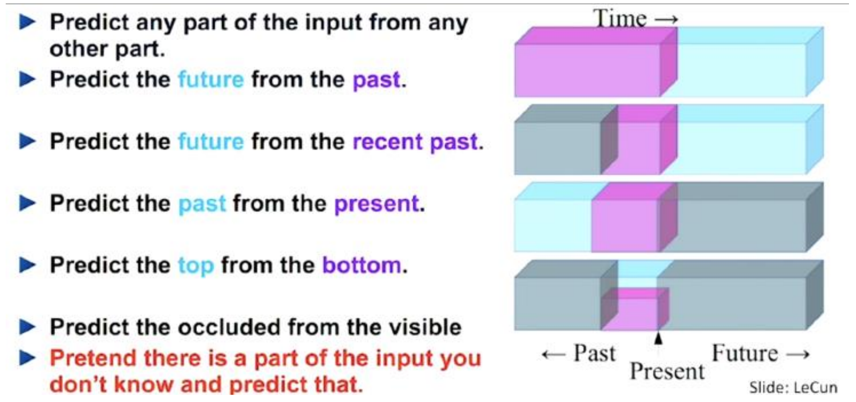


Figure source: Dive into Deep Learning (Zhang et al, 2023)

Unsupervised Pre-training

- Self-supervised learning is a powerful idea to learn good representations unsupervisedly


Self-supervised learning will help us learn a good encoder (feature representation)
Hide part of the input and predict it using the remaining parts
Encoder
Encoder Features
Channel-wise Fully Connected
Decoder Features
Decoder
$\mathcal{L}$



- Predict any part of the input from any other part.
- Predict the future from the past.
- Predict the future from the recent past.
- Predict the past from the present.
- Predict the top from the bottom.
- Predict the occluded from the visible
- Pretend there is a part of the input you don't know and predict that.

- Self-supervised learning is key to unsupervised pre-training of deep learning models
  - Such pre-trained models can be fine-tuned for any new task given labelled data
- Models like BERT, GPT are usually pre-trained using self-supervised learning
  - Then we can finetune them further for a given task using labelled data for that task

Auto-encoders

A special type of self-supervised learning: The whole input is being predicted by first compressing it and then uncompressing

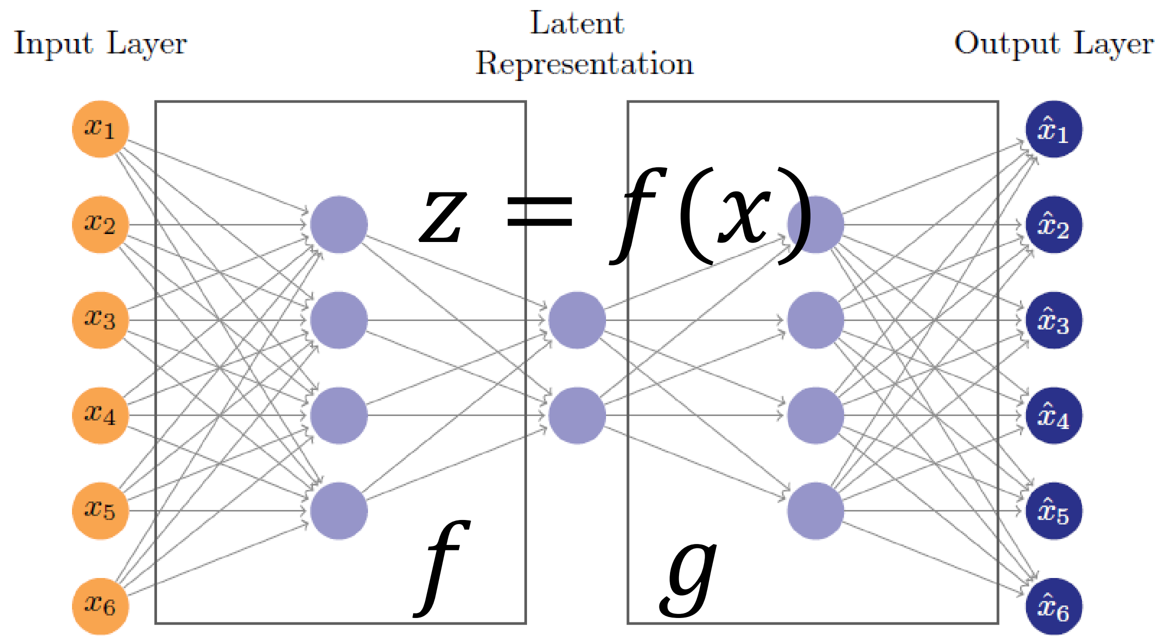
- Auto-encoders (AE) are used for unsupervised feature learning
- Consist of an encoder f and a decoder g
  - f and g can be deep neural networks (MLP, RNN, CNN, etc)

Note: Usually only the encoder is of use after the AE has been trained (unless we want to use the decoder for reconstructing the inputs later)

If using a prior on $\boldsymbol{z}$, we can a probabilistic latent variable model called variational auto-encoder (VAE)

VAE can also generate synthetic data usings its decoder (standard AE's decoder can't generate "new" data)

$$\hat{x} = g(f(x))$$



Goal: Learn f and g s.t. $\|x - \hat{x}\|$ is small

Dimensionality of $\boldsymbol{z}$ can be chosen to be smaller or larger than that of $\boldsymbol{x}$

If using AE for dimensionality reduction

Sometimes we want to learn "overcomplete" feature representations of the input

In such cases, need to impose additional constraints on f so that we don't learn an identify mapping from $\boldsymbol{x}$ to $\boldsymbol{z}$

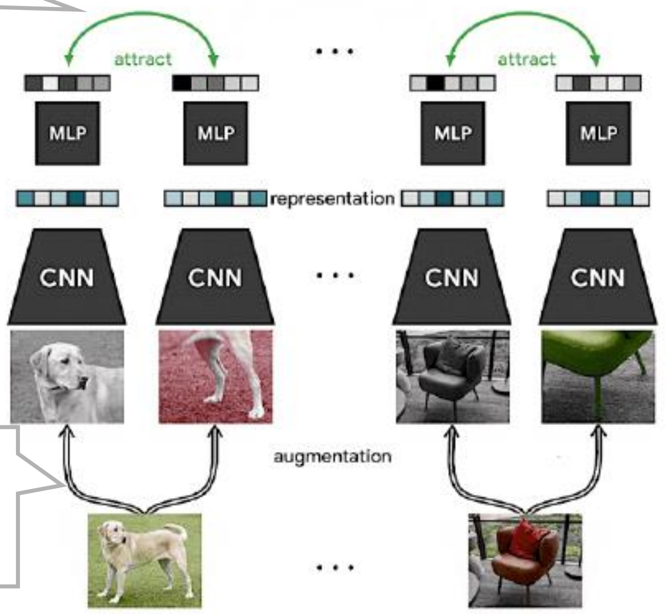
Contrastive Learning

Or "triplets" (e.g., "cat" is more similar to "dog" than to a "table")

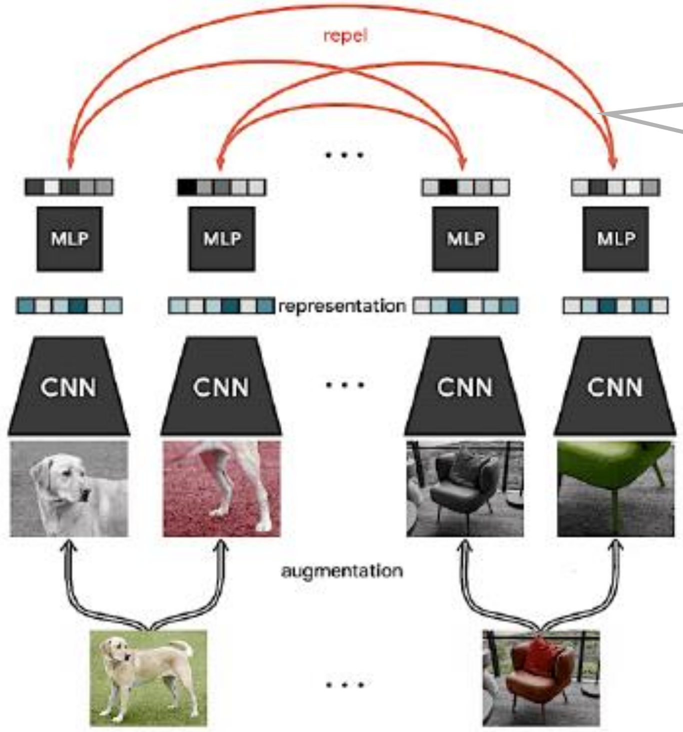
- Can learn good features by comparing/"contrasting" similar and dissimilar object pairs
- Such pairs can be provided by to the algorithm (as supervision), or the algorithm can generate such pairs by itself using "data augmentation" (as shown in example below)

The corresponding embeddings for such pairs must be close ("attract") to each other

Augmentation by cropping and resizing. The class of the image remains unchanged in this augmentation

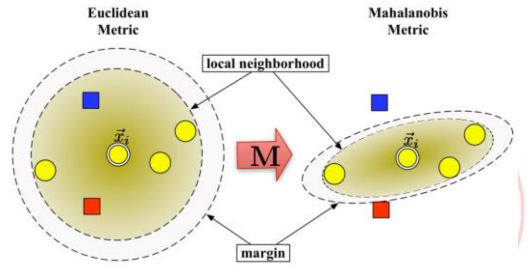


The embeddings for "non-match" pairs must be far away ("repel") from each other



Distance metric learning by learning **M** given similar/dissimilar pairs

$$d(\boldsymbol{x}_i, \boldsymbol{x}_j) = \sqrt{(\boldsymbol{x}_i - \boldsymbol{x}_j)^\top \mathbf{M}(\boldsymbol{x}_i - \boldsymbol{x}_j)}$$



- Such "contrastive learning" of features is also related to "distance metric learning" algos

Bias-Variance Trade-off

- Assume $\mathcal{F}$ to be a class of models (e.g., linear classifiers with some pre-defined features)
- Suppose we've learned a model $f \in \mathcal{F}$ learned using some (finite amount of) training data
- We can decompose the test error $\epsilon(f)$ of f as follows

E.g., going from linear models to deep nets or by adding more features

Can bias reduce by making class $\mathcal{F}$ richer

$$\epsilon(f) = \underbrace{\left[\min_{f^* \in \mathcal{F}} \epsilon(f^*)\right]}_{\text{approximation error}} + \underbrace{\left[\epsilon(f) - \min_{f^* \in \mathcal{F}} \epsilon(f^*)\right]}_{\text{estimation error}}$$

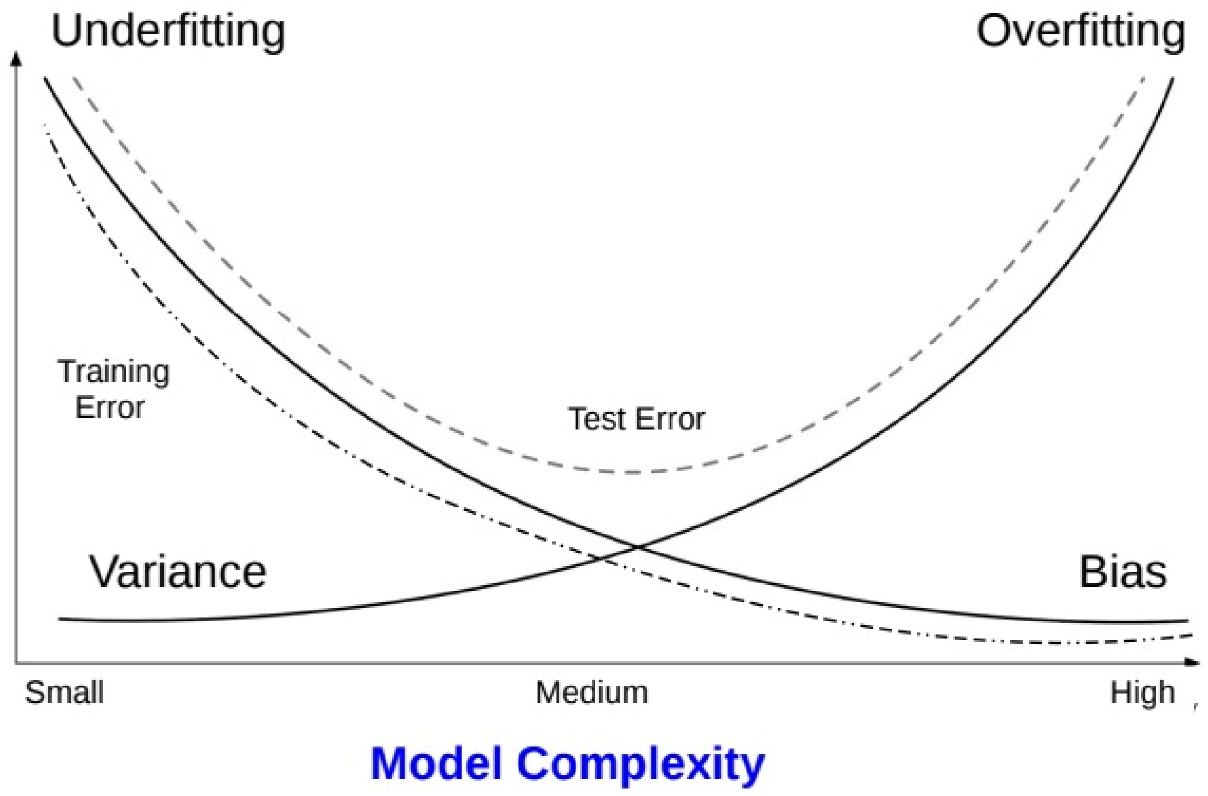
Reason: We are now learning a more complex model using the same amount of training data

Making $\mathcal{F}$ richer will also cause estimation error to increase

- Here f^* is the best possible model in $\mathcal{F}$ assuming infinite amount of training data
- Approximation error: Error of f^* because of model class $\mathcal{F}$ being too simple
  - Also known as "bias" (high if the model is simple)
- Estimation error: Error of f (relative to f^*) because we only had finite training data
  - Also known as "variance" (high if the model is complex)
- Because we can't keep both low, this is known as the bias-variance trade-off

Bias-Variance Trade-off

- Bias-variance trade-off implies how training/test losses vary as we increase model complexity



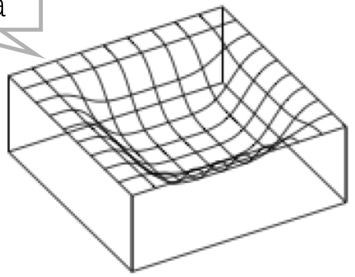
Deep Neural Nets and Bias-Variance Trade-off

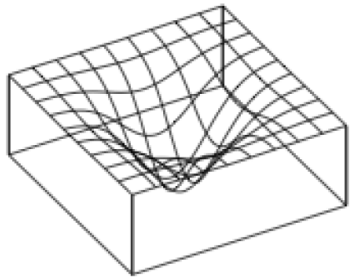
- Bias-variance trade-off doesn't explain well why deep neural networks work so well
  - They have very large model complexity (massive number of parameters – massively "overparametrized")

- Despite being massively overparametrized, deep neural nets still work well because
  - Implicit regularization: SGD has noise (randomly chosen minibatches) which performs regularization
  - These networks have many local minima and all of them are roughly equally good
  - SGD on overparametrized models usually converges to "flat" minima (less chance of overfitting)

Such a solution is less likely to be an overfitted solution because other nearby solutions are also reasonably good

A flat minima





A sharp minima

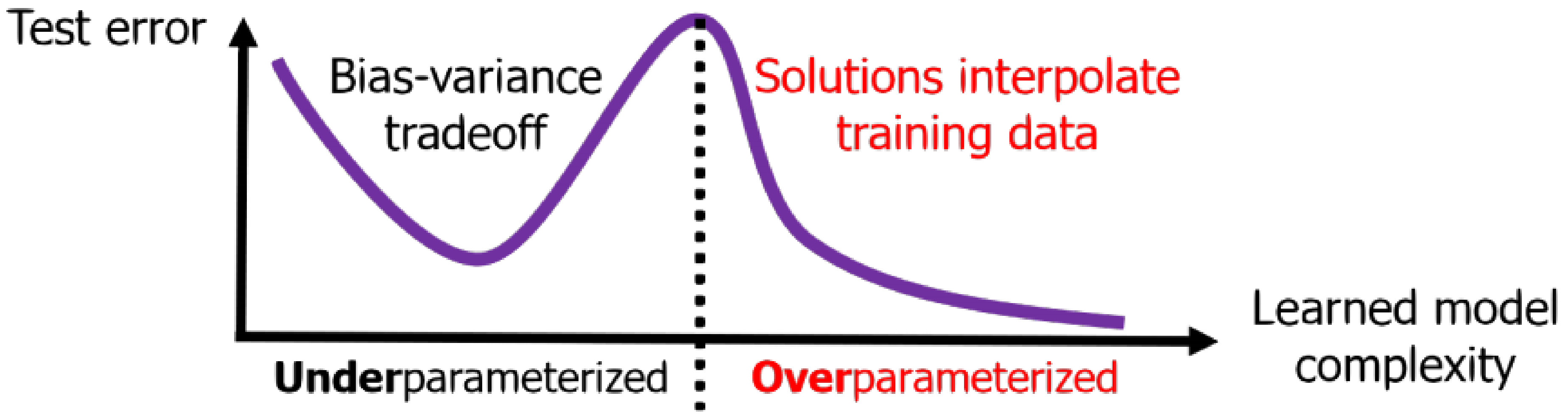
Such minima are not good because they might represent an overfitted solution

SGD because of the "noise" can escape such sharp minima

  - Learning of good features from the raw data
  - Ensemble-like effect (a deep neural net is akin to an ensemble of many simpler models)
  - Trained on very large datasets

Double Descent Phenomenon

- Overparametrized deep neural networks exhibit a "double descent" phenomenon



- Bias-variance trade-off seen only in the underparametrized regime
- Beyond a point (in the overparametrized regime), the test error starts decreasing once again even as the model gets more and more complex

Figure source: "A Farewell to the Bias-Variance Tradeoff? An Overview of the Theory of Overparameterized Machine Learning" (Dar et al, 2021)

Debugging ML Algorithms

What is going wrong?

- What to do when our model (say logistic regression) isn't doing well on test data
  - Use more training examples?
  - Use a smaller number of features?
  - Introduce new features (can be combinations of existing features)?
  - Try tuning the regularization parameter?
  - Run (the iterative) optimizer longer, i.e., for more iterations
  - Change the optimization algorithm (e.g., GD to SGD or Newton..) or the learning rate?
  - Give up and switch to a different model (e.g., SVM or deep neural net)?

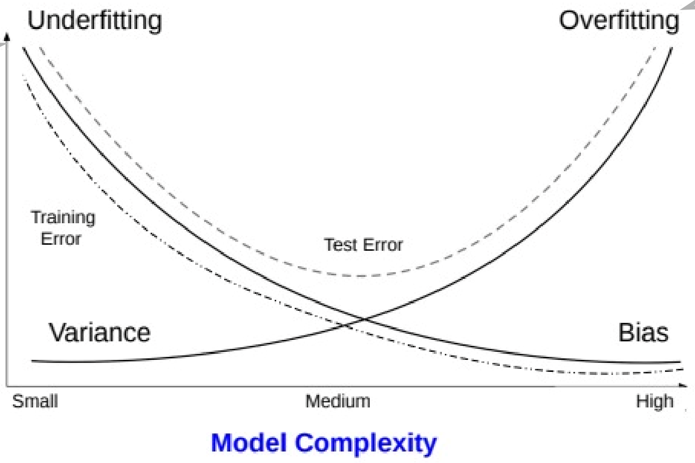
High-Bias or High-Variance?

- The bad performance (low accuracy on test data) of a model could be due either
  - High Bias: Too simple model; doesn't even do well on training data
  - High Variance: Even small changes in training data lead to high fluctuation in model's performance
- High bias means underfitting, high variance means overfitting
- Looking at the training and test error can tell which of the two is the case

Both training and test errors are large

Adding more training examples won't help as the model itself is too simple

Use a more complex model or increase the number of features



Small training error and large test error

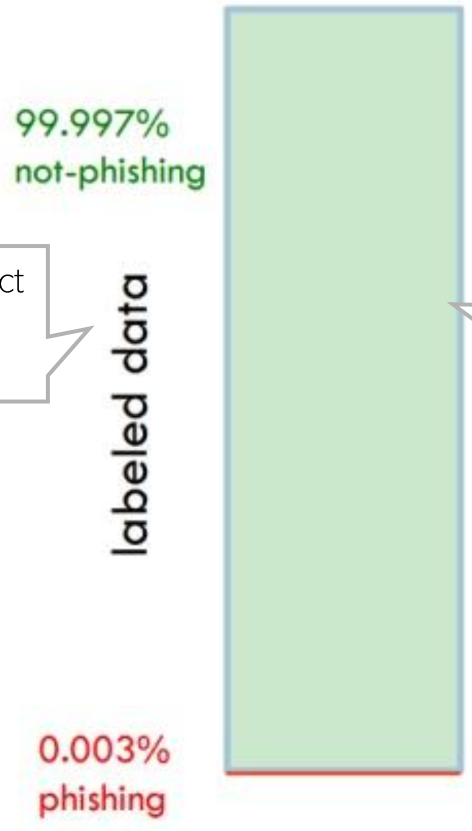
Adding more training examples can possibly help here because we possibly have a complex model but not enough training data to learn it well

Or regularize the model better to prevent overfitting or use fewer features, or switch to a simpler model

Learning from Imbalanced Data

Learning when classes are imbalanced

- When classes are imbalanced, even a “stupid” classifier can give high accuracy but looking at accuracy alone may be misleading



A stupid classifier: Simply predict everything as non-phishing. Gives close to 100% accuracy

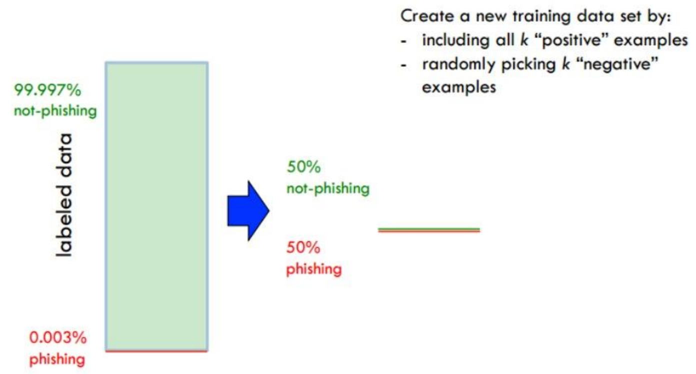
But likely to do badly on test data, particularly phishing mails (all will be predicted as non-phishing)

If we use classification loss/accuracy as our loss function, the learned model may indeed prefer simply predicting each input to be from the majority class

For such data with imbalanced classes, we need ways so that the majority class is not given undue importance

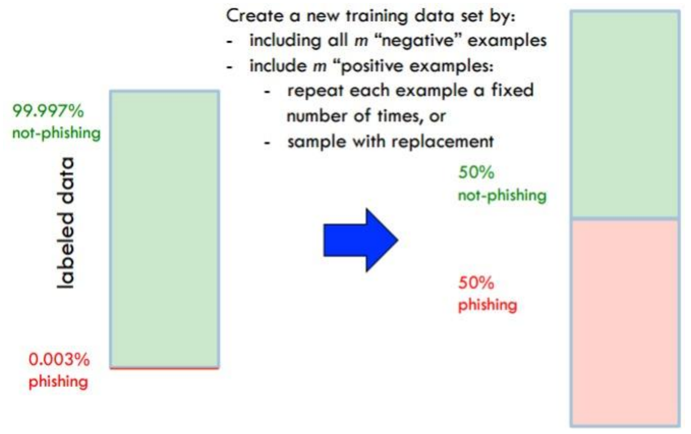
Solution 1: Balancing the training data

- Can balanced the training data by
  - Under-sampling the majority class examples



Create a new training data set by:
- including all *k* "positive" examples
- randomly picking *k* "negative" examples

  - Over-sampling the minority class examples



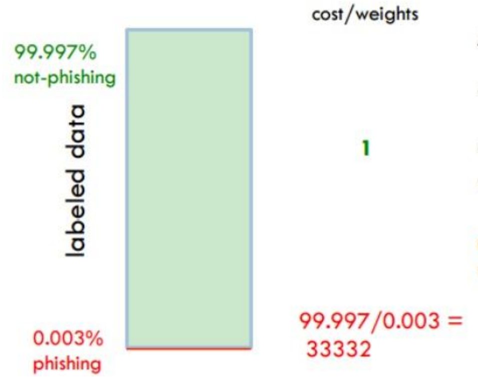
Create a new training data set by:
- including all *m* "negative" examples
- include *m* "positive examples:
  - repeat each example a fixed number of times, or
  - sample with replacement

Equivalent to

Weighted loss function with much larger importance given to loss function terms of positive examples than negative examples

$\mathcal{L}(\boldsymbol{w}) = \sum_{i=1}^{N} \beta_{y_i} \ell(\boldsymbol{x}_i, y_i, \boldsymbol{w})$

where $\beta_{+1} \gg \beta_{-1}$



Add costs/weights to the training set

"negative" examples get weight 1

"positive" examples get a much larger weight

change learning algorithm to optimize weighted training error

Solution 2: Changing the loss function

- Don't use loss functions that define loss or accuracy on per-example basis

This loss function is a simple sum of losses on individual training examples. Not ideal for imbalanced classes

$$L(\boldsymbol{w}) = \sum_{i=1}^{N} \ell(x_i, y_i, \boldsymbol{w})$$

- Instead, use loss function that use example pairs (one positive and one negative)
- Assuming our model to be defined by some function $f(\boldsymbol{x})$ (e.g., $\boldsymbol{w}^\top \boldsymbol{x}$), define a loss

An input with positive label

An input with negative label

$$\ell(f(\mathbf{x}_n^+), f(\mathbf{x}_m^-)) = \begin{cases} 0, & \text{if } f(\mathbf{x}_n^+) > f(\mathbf{x}_m^-) \\ 1, & \text{otherwise} \end{cases}$$

Now we don't care about per-example accuracy but care about whether the positive examples get a higher score than the negative examples (i.e., we are only preserving their relative rank)

Usual regularaizer on f

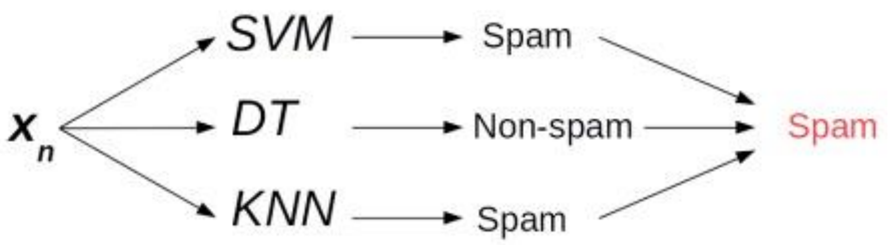
$$\sum_{n=1}^{N_+} \sum_{m=1}^{N_-} \ell(f(\mathbf{x}_n^+), f(\mathbf{x}_m^-)) + \lambda R(f)$$

Such loss functions can known as "pairwise loss functions"

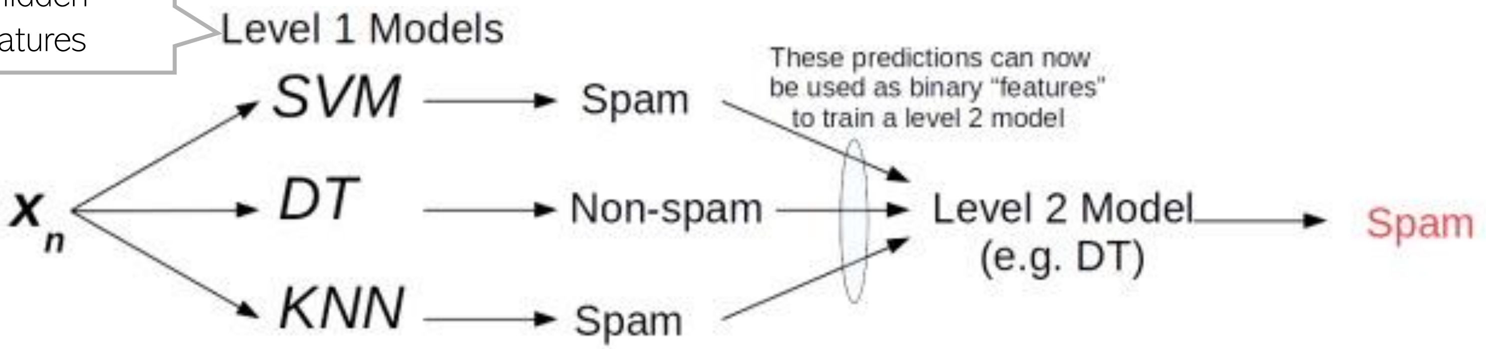
Ensemble Methods

Some Simple Ensembles

- Voting or Averaging of predictions of multiple models trained on the same data



- "Stacking": Use predictions of multiple already trained models as "features" to train a new model and use the new model to make predictions on test data


Level 1 Models
x_n
SVM
DT
KNN
Spam
Non-spam
Spam
These predictions can now be used as binary "features" to train a level 2 model
Level 2 Model (e.g. DT)
Spam

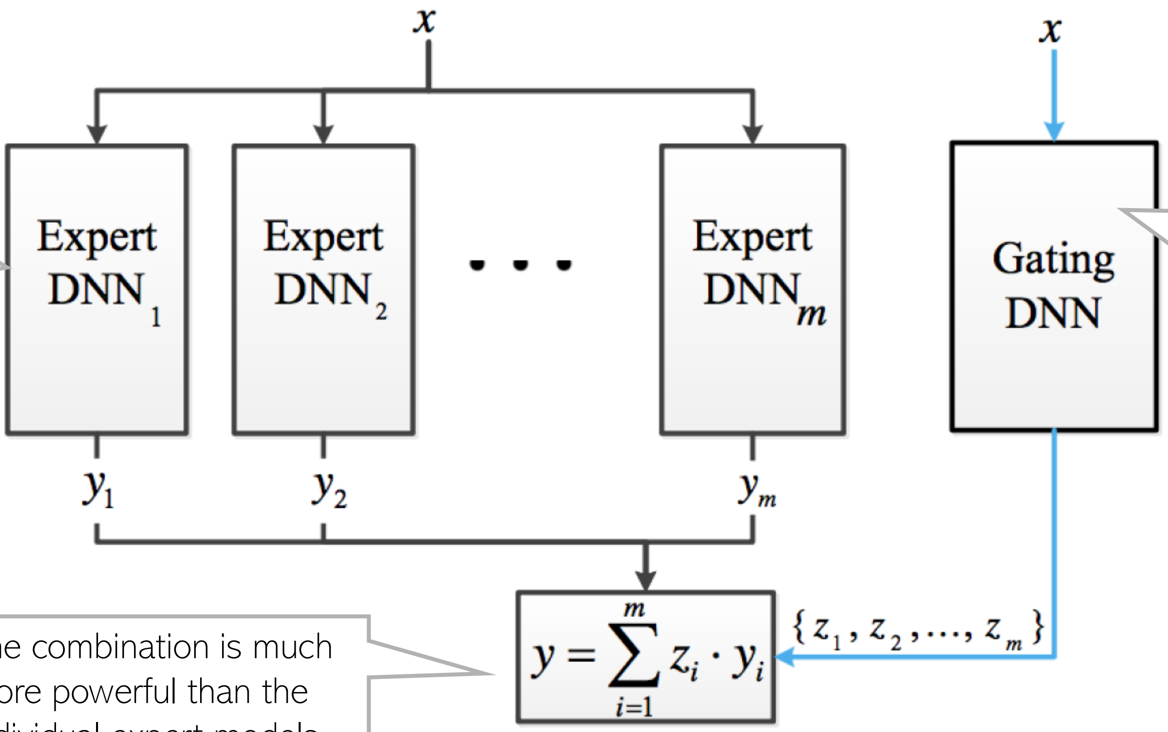
Stacking sort of has a flavor of deep learning where hidden layers extract good features

Here, that role is being played by these other already trained models

In stacking, level 1 and level 2 models are trained independently (first level 1 and then level 2)

Mixture of Experts (MoE) based Ensemble

- Mixture of Experts (MoE) is a very general idea
- We assume $\boldsymbol{m}$ "simple" models, usually of the same type, e.g., $\boldsymbol{m}$ linear SVMs or $\boldsymbol{m}$ logistic regression models, or $\boldsymbol{m}$ deep neural nets (usually all with same architecture)



Experts models and the gating/routing function are learned jointly using all the training data

MoE can also be formulated like a latent variable model and learned using ALT-OPT or EM

The combination is much more powerful than the individual expert models

Gating (also called "routing") function which outputs an $\boldsymbol{m}$ dimensional vector $\boldsymbol{z} = [z_1, z_2, \ldots, z_m]$ which indicates us how much importance should be given to each expert for this input $\boldsymbol{x}$

Some variants of MoE pick only one expert or a small subset of expert

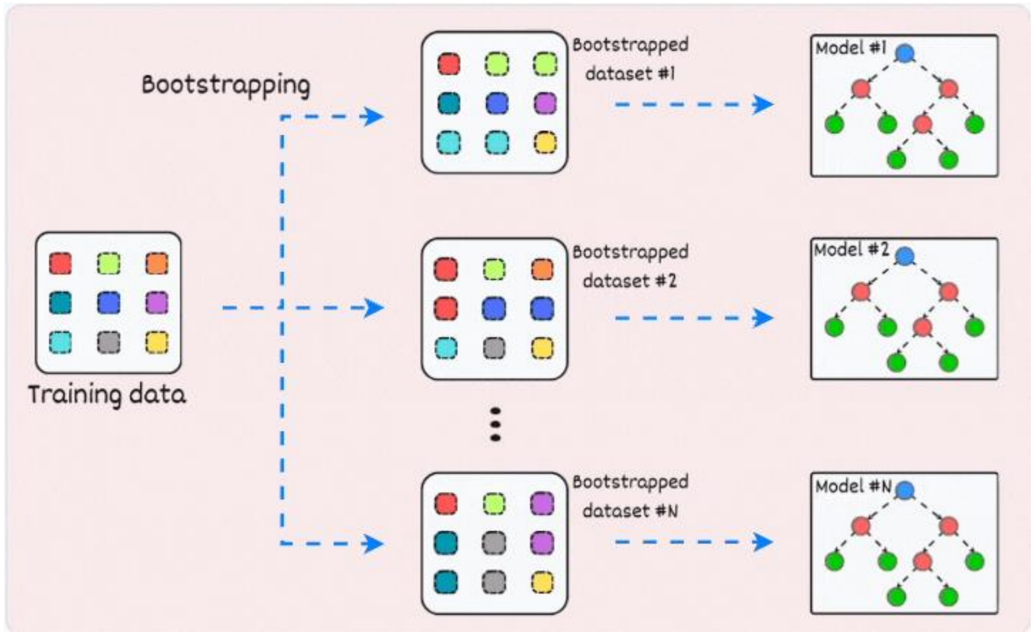
In that case $\boldsymbol{z}$ will be a one-hot or sparse binary vector

- MoE is very popular in classical ML as well as "modern" deep learning

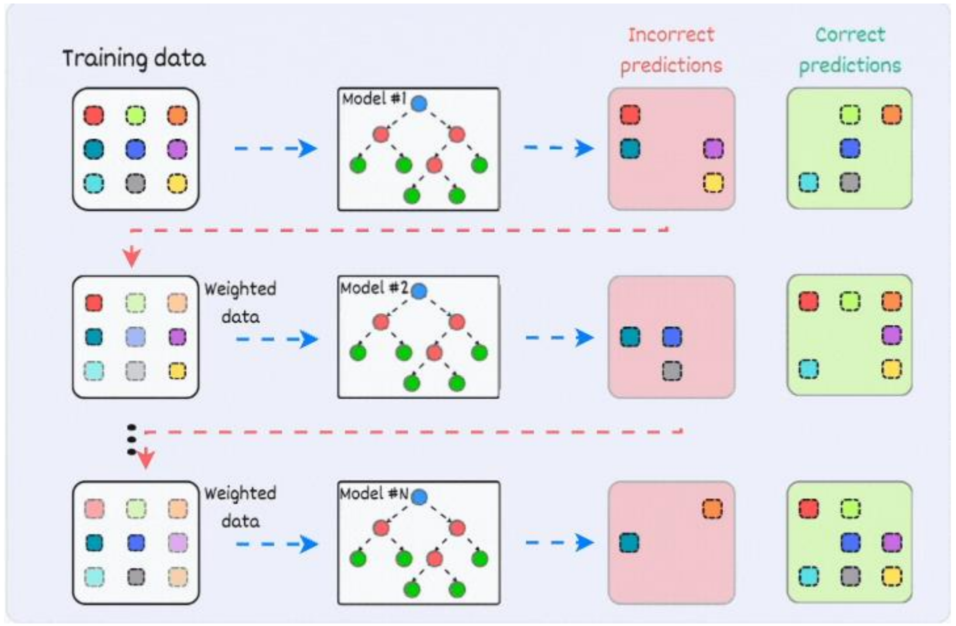
Ensembles using Bagging and Boosting

- Both use a single training set $\mathcal{D}$ to learn an ensemble consisting of several models
- Both construct N datasets from the original training set $\mathcal{D}$ and learn N models

Bagging



Boosting



- Bagging can do this in <u>parallel</u> for all the N models
- Boosting requires a <u>sequential</u> approach for N rounds

Figure credit: https://www.blog.dailydoseofds.com/p/an-animated-guide-to-bagging-and

Bagging (Bootstrap Aggregation)

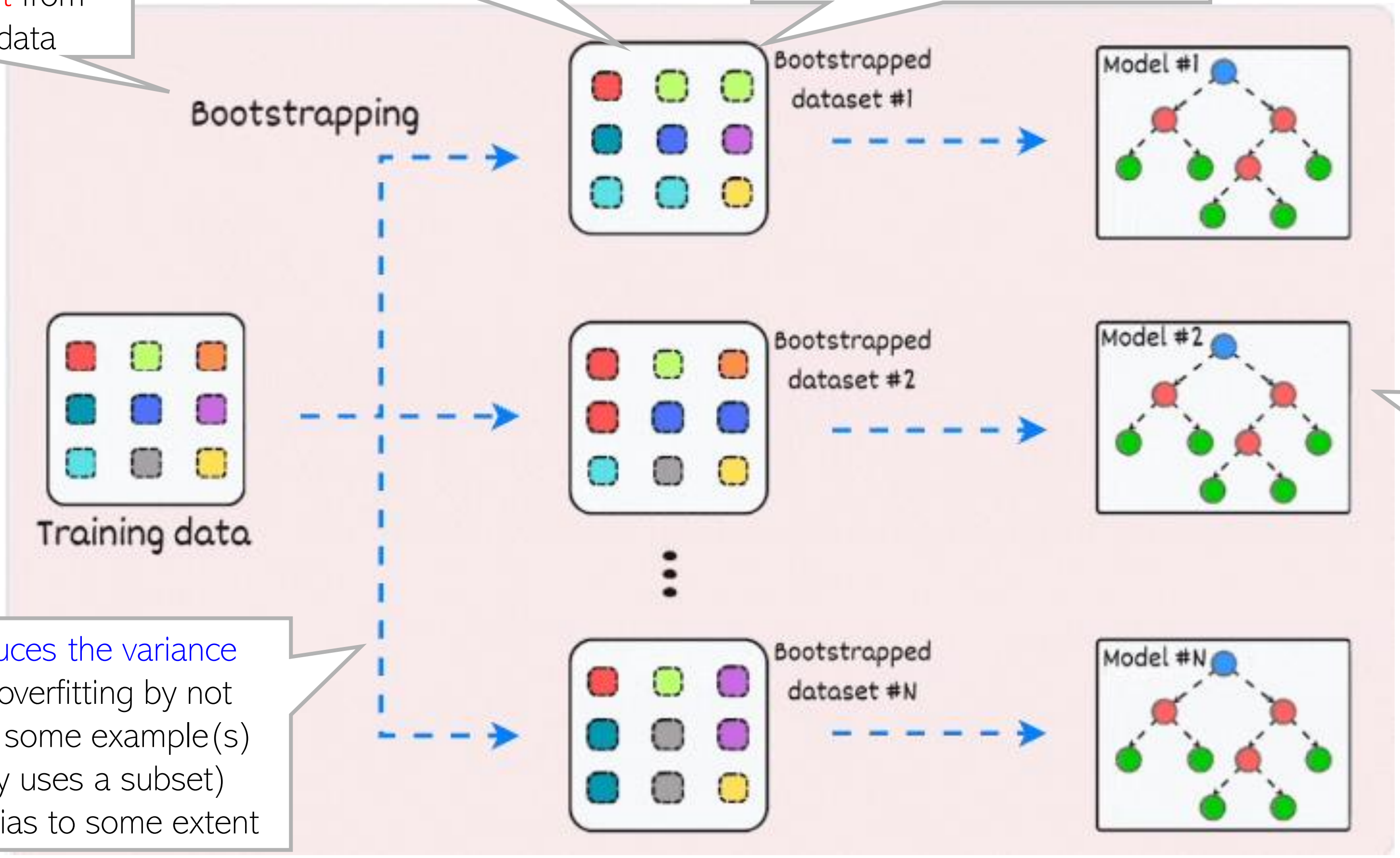
Generated bootstrapped datasets using sampling with replacement from original training data

Each bootstrapped dataset consists of a subset of all the training examples

Sampling with replacement will give about 63% unique training examples (if both datasets are of equal size)

Bootstrapping can also be done at feature level (each model using a randomly chosen subset of all the features)

Random forests use both bootstrapping at examples level and at features level



To predict the output for a new test input, we apply all these models and average their predictions

Bagging primarily reduces the variance (reduces chances of overfitting by not relying completely on some example(s) since each model only uses a subset) but can also reduce bias to some extent

Figure adapted from: https://www.blog.dailydoseofds.com/p/an-animated-guide-to-bagging-and

Boosting

Boosting assumes that the individual models are simple/weak models that can be easily learned

Boosting trains them sequentially and combines them to get a "boosted" powerful model

Note that here we have two types of importances: importance of each training example and importance of each model

"Weighted data" means that we are increasing the importance of examples that were mis-predicted in the previous round and decrease it for examples that were correctly predicted

Importance of each model is its accuracy

The final model will be an important-weighted combination of all these models

Boosting primarily reduces the bias by making the weak (underfitted) models stronger but can also reduce variance to some extent

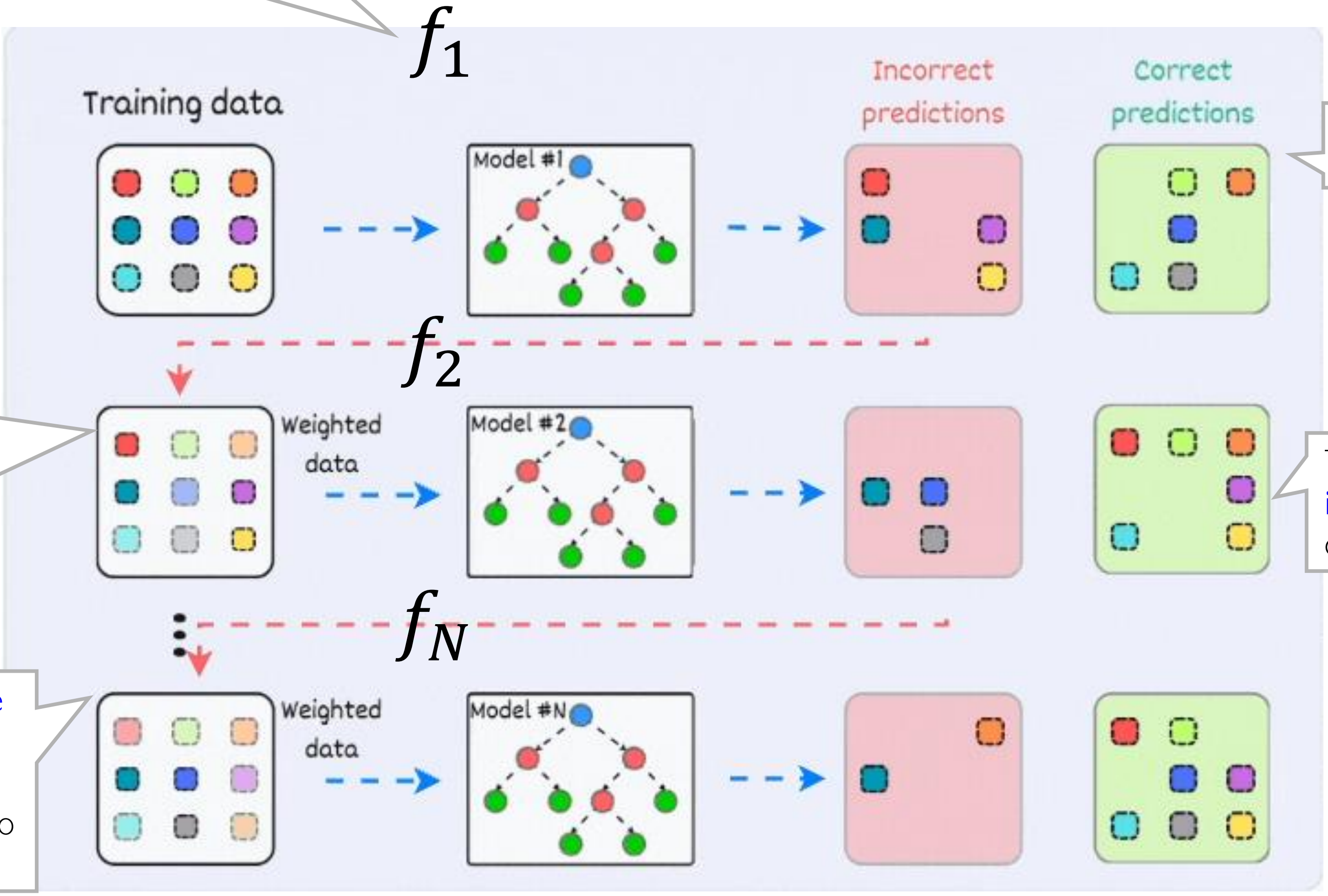


Figure adapted from: https://www.blog.dailydoseofds.com/p/an-animated-guide-to-bagging-and

A Boosting Algo: AdaBoost (Adaptive Boosting)

Importance of the training example $(\boldsymbol{x}_i, y_i)$

We might know this beforehand or estimate it during training

- In many ML problems, we can assign importance weight to each example, e.g., by weighing each term in the loss functions, i.e., $\mathcal{L}(\boldsymbol{w}) = \sum_{i=1}^{N} \beta_i \ell(\boldsymbol{x}_i, y_i, \boldsymbol{w})$
- AdaBoost is based on optimizing such a loss function

Initially assume equal importance for all training examples

  - Initialize the ensemble as $\mathcal{E} = \{\}$ and $\boldsymbol{\beta}$ as $\boldsymbol{\beta}^{(0)} = [\frac{1}{N}, \frac{1}{N}, \dots, \frac{1}{N}]$
  - For round $t = 1,2,\dots,T$
    - $\boldsymbol{w}^{(t)} = \operatorname{argmin}_{\boldsymbol{w}} \sum_{i=1}^{N} \beta_i^{(t-1)} \ell(\boldsymbol{x}_i, y_i, \boldsymbol{w})$ and add it to ensemble $\mathcal{E} = \{\mathcal{E} \cup \boldsymbol{w}^{(t)}\}$
    - Define the total loss of $\boldsymbol{w}^{(t)}$ as $L(\boldsymbol{w}^{(t)}) = \sum_{i=1}^{N} \beta_i^{(t-1)} \ell(\boldsymbol{x}_i, y_i, \boldsymbol{w}^{(t)})$

Or the importance weighted total error

    - Compute the "importance" of $\boldsymbol{w}^{(t)}$ for the $\mathcal{E}$ as $\alpha_t = f(L(\boldsymbol{w}^t))$

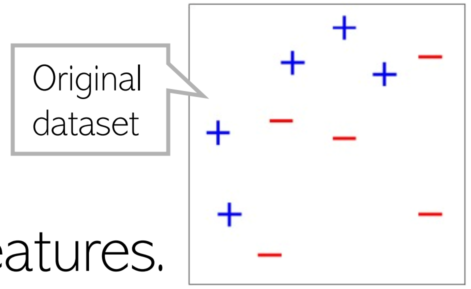
f is some function such that α_t is high if total loss $L(\boldsymbol{w}^t)$ is low, and vice-versa

    - Increase/decrease importance β_i of each training instance $(\boldsymbol{x}_i, y_i)$ for next round as

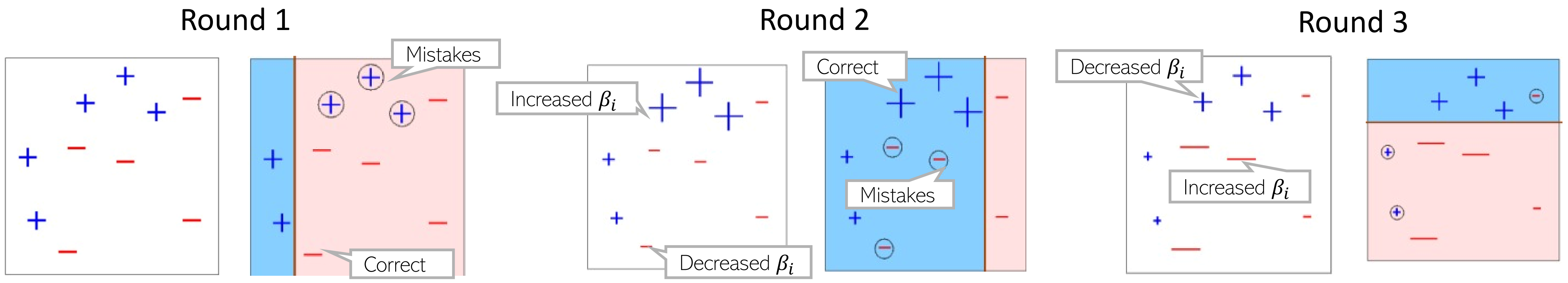
$$\beta_i^{(t)} \propto \begin{cases} \beta_i^{(t-1)} \times \exp\left(\alpha_t \ell\left(\boldsymbol{x}_i, y_i, \boldsymbol{w}^{(t)}\right)\right) & \text{(Increase if } \boldsymbol{w}^{(t)} \text{ mispredicted } (\boldsymbol{x}_i, y_i)) \\ \beta_i^{(t-1)} \times \exp(-\alpha_t \ell(\boldsymbol{x}_i, y_i, \boldsymbol{w}^{(t)})) & \text{(Decrease if } \boldsymbol{w}^{(t)} \text{ correctly predicted on } (\boldsymbol{x}_i, y_i)) \end{cases}$$

  - Final model is $\hat{\boldsymbol{w}} = \sum_{t=1}^{T} \alpha_t \boldsymbol{w}^{(t)}$ importance-weighted average of all $\boldsymbol{w}^{(t)}$'s

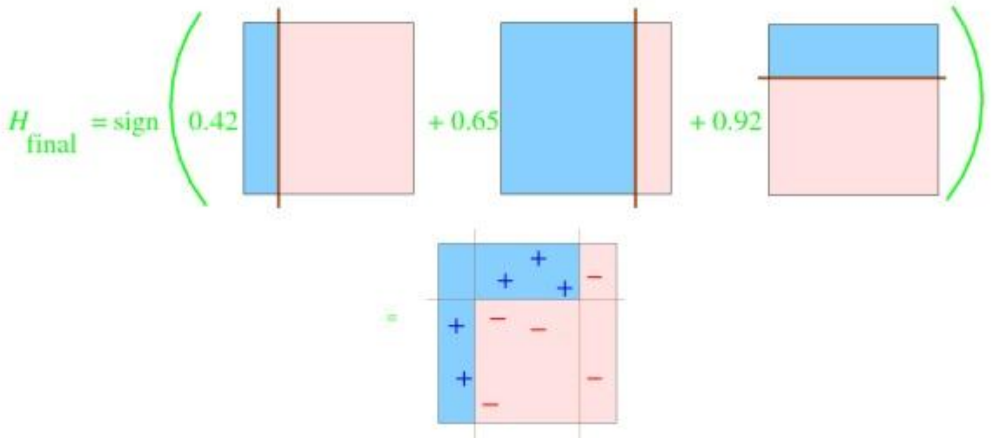
AdaBoost: An Illustration



- Suppose we have a binary classification problems with each input having 2 features.
- Suppose we have a weak model like a simple DT (decision stump)
- Illustration of AdaBoost using a decision stump if run for 3 rounds



- The ensemble represents the overall model
- We got a nonlinear model from 3 simple linear models
- Note that the ensemble was constructed sequentially



Gradient Boosting

- Consider learning a function $f(x)$ by minimizing a squared loss $\frac{1}{2}\left(y - f(x)\right)^2$
- Gradient boosting is a sequential way to construct such $f(x)$
- For simplicity, assume we start with $f_0(x) = \frac{1}{N}\sum_{i=1}^{N} y_i$
- Given previously learned model $f_m(x)$, let's assume the following "improvement" to it

$$f_{m+1}(x) = f_m(x) + h(x)$$

"Residual" which, if added to $f_m(x)$, will make the new prediction $f_{m+1}(x)$ closer to y

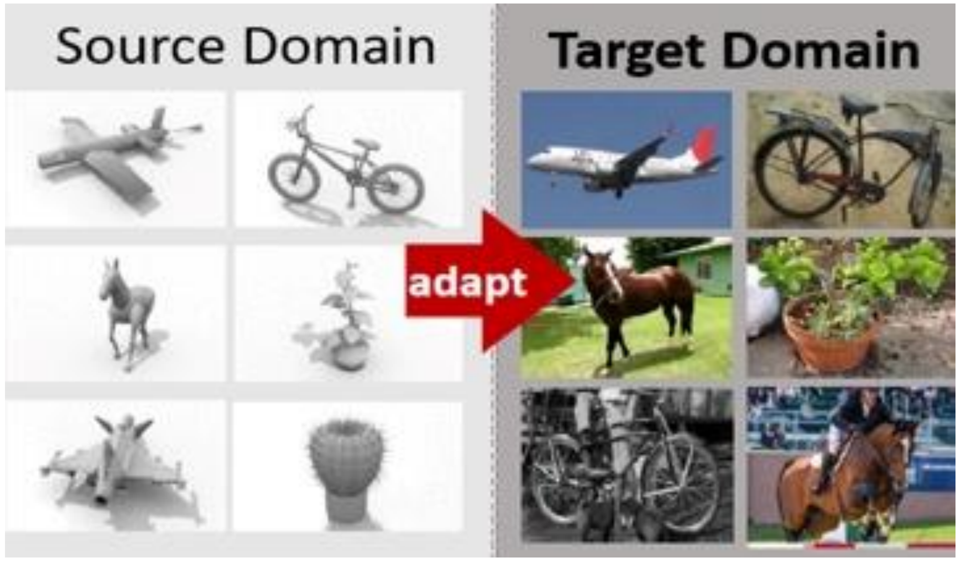
- Thus the goal for the next round is to learn the "residual" $h(x) = y - f_m(x)$
- Residual is negative gradient of the loss w.r.t. $f(x)$ - thus called "gradient boosting"
- The final model $f_M(x)$, once the residual is sufficiently small, is what we will use
- The idea of gradient boosting is applicable to classification too
- XGBoost (eXtreme Gradient Boosting) is a very popular grad boosting algo

Based on sequentially constructing a DT

Domain Adaptation

- We may have a "source" model trained on data from some domain
- We might want to deploy it in a new domain
- Performance of the source model will suffer
- To prevent this, we usually perform "domain adaptation" or "transfer learning"
- These are broad terms covering a variety of techniques that "finetune" the source model using labelled/unlabeled data from the new domain

We do expect some "commonality" (e.g., some common set of features) between the two domains otherwise we can't hope to have any adaptation/transfer



The ending note..

- Good features are important for learning well
- The "classical" ML methods we studied in this course still continue to have high relevance
- Success of deep learning is largely attributed to (automatically learned) good features
- Deep learning is not a panacea – often simple classical models can do comparably/better
- First understand your data (plot/visualize/look at some statistics of the data, etc)
- Always start with a simple model that you understand well
  - Try to first understand if your data really needs a complex model
- Think carefully about your features, how you compute similarities, etc.
- Helps to learn to first diagnose a learning algorithm rather than trying new ones
  - Understanding of optimization algos, loss function, bias-variance trade-offs, etc is important
- No free lunch. No learning algorithm is "universally" good