GSAS @NIDA : Deep Learning

MLP 2

Assoc.Prof.Thitirat Siriborvornratanakul, Ph.D.

Email: thitirat@as.nida.ac.th
Website: http://as.nida.ac.th/~thitirat/

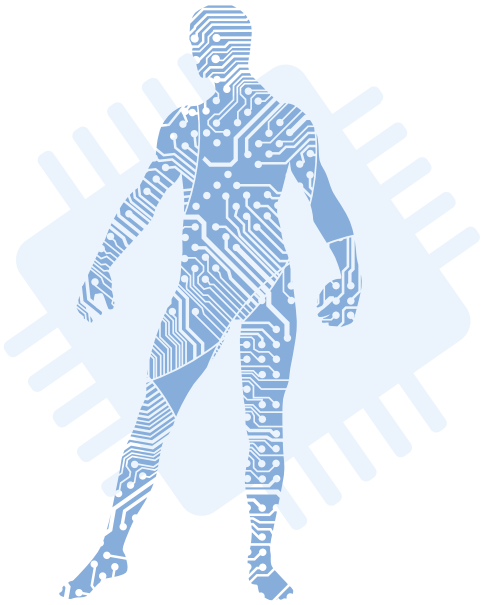
1

OUTLINE

01 Problem 1
Unstable gradients in deep neural networks and how to deal with it

02 Problem 2
Overfitting in deep neural networks and how to solve it

03 Coding
Combine everything and implement a program with PyTorch

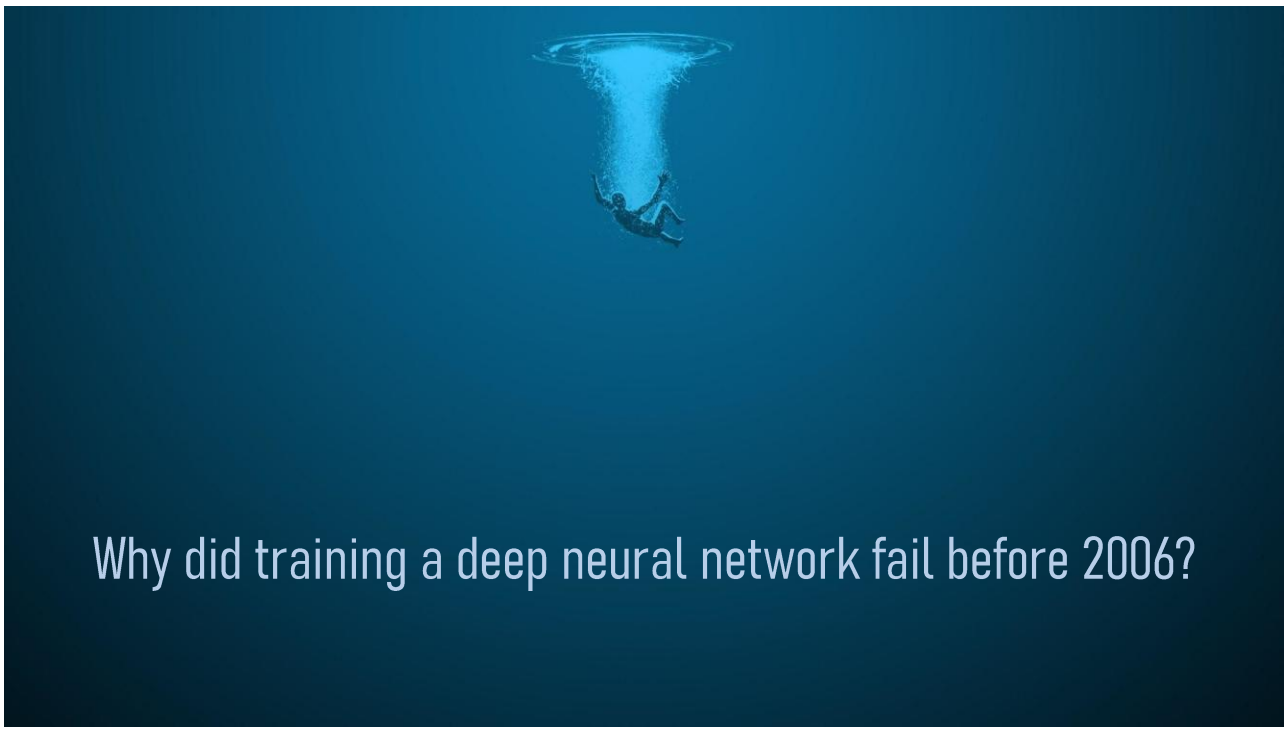


2

Problem 1

Unstable gradients in deep neural networks and how to deal with it

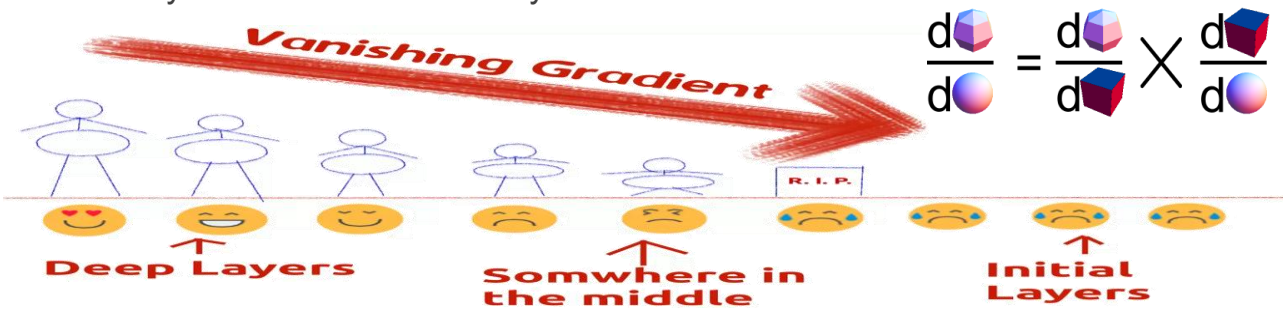
3



4

Gradient Vanishing

- In the early 2000s:
  - Gradients get smaller and smaller as the algorithm progresses down to the lower layers.
  - As a result, the Gradient Descent update leaves the lower layers' connection weights virtually unchanged, and training never converges to a good solution.
- **Gradient vanishing** was one of the reasons deep neural networks were mostly abandoned in the early 2000s.



5

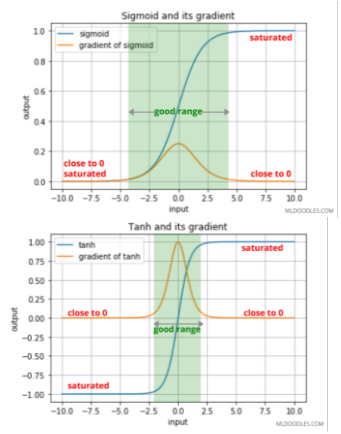
Problem of unstable gradients

- **Gradient vanishing:**
  - Slow training progress
  - Low weight updates
  - Dead neurons: Neurons in some layers have very small or zero outputs, causing them to become inactive.
- **Gradient exploding:** Large updates to weights (= large gradients) during training that cause a numerical overflow or underflow.
  - Large weight updates
  - Null, NaN, or Inf values in weights
  - Oscillating performance: The model's accuracy fluctuates over time, indicating instability.
- How to detect unstable gradients?
  - Monitor the magnitude of gradients (whether they are too small or too large) during the training process using a real-time visualization tool (e.g., TensorBoard)

6

Tackle unstable gradients

- The work of Glorot and Bengio (2010) found a few suspects to the unstable gradient when training a deep neural network.
  - **Nonsaturating activation functions:**
    - Other activation functions (e.g., ReLU) behave much better in deep neural networks than sigmoid. This is because ReLU does not saturate for positive values.
  - **Weight initialization:**
    - It took over a decade for researchers to realize how important this trick is.
    - Using Glorot initialization can speed up training considerably, and it is one of the tricks that led to the success of Deep Learning.



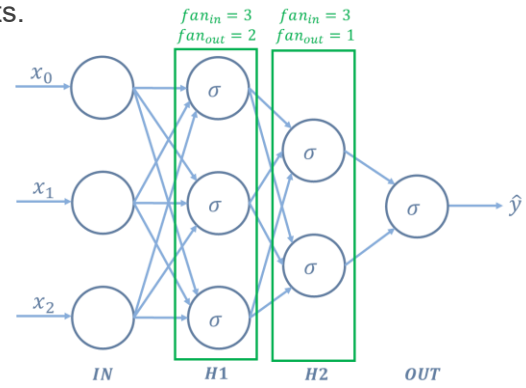
Xavier Glorot and Yoshua Bengio, "Understanding the difficulty of training deep feedforward neural networks," In Procs. of Machine Learning Research, pp.249-256, 2010.

7

Initialization strategy

- A too-large initialization leads to exploding gradients.
- A too-small initialization leads to vanishing gradients.

- For signal to flow properly in both forward and backward directions without dying out, exploding or saturating, we need the gradients to have equal variance before and after flowing through a layer in the reverse direction.
  - The mean of the activations should be zero.
  - The variance of the activations should stay the same across every layer.

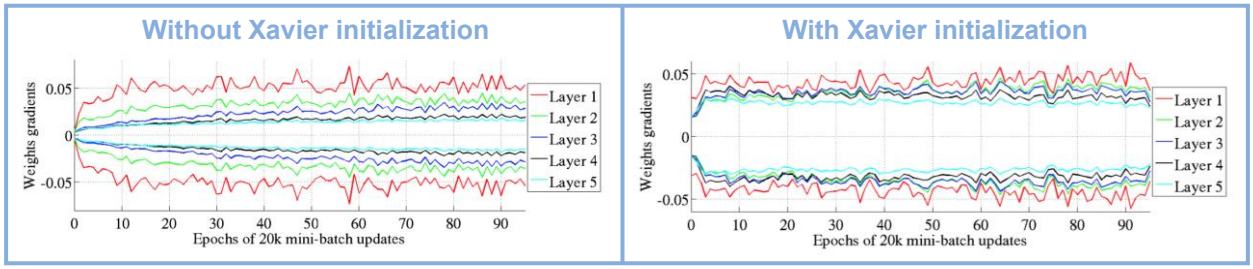


- Unfortunately, it is not possible to guarantee both unless the layer has an equal number of incoming network connections (fan_{in}) and outgoing network connections (fan_{out}).

8

Initialization strategy

- Xavier initialization (a.k.a., Glorot initialization) proposed a good compromise that has proven to work very well in practice, just setting a layer's weights to values chosen from a random uniform distribution that's bounded between $\left[-\sqrt{\frac{6}{fan_{in}+fan_{out}}}\ ,\ \sqrt{\frac{6}{fan_{in}+fan_{out}}}\right]$



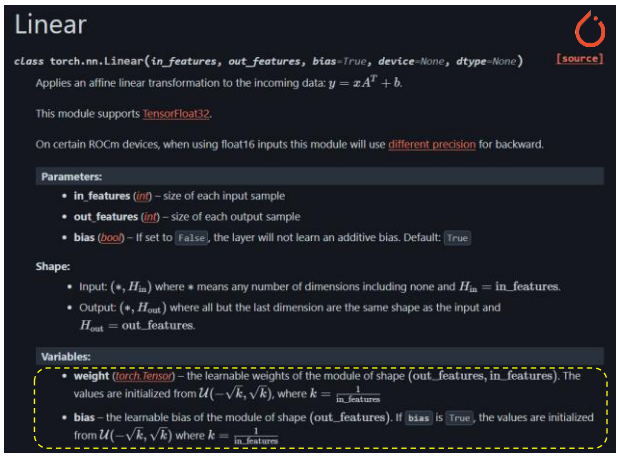
Xavier Glorot and Yoshua Bengio, "Understanding the difficulty of training deep feedforward neural networks," In Procs. of Machine Learning Research, pp.249-256, 2010.

9

Initialization strategy

- PyTorch uses LeCun initialization by default. This can be changed by accessing the layer's weights directly and applying an initialization function from the `torch.nn.init` module. For example, `torch.nn.init.xavier_uniform_(model.fc2.weight)`.

Strategy	Activation functions	Uniform distribution $[-A, A]$	Normal distribution (with mean = 0)
Glorot, Xavier (2010)	None, tanh, sigmoid, softmax	$A=\sqrt{\frac{6}{fan_{in}+fan_{out}}}$	$\sigma=\sqrt{\frac{2}{fan_{in}+fan_{out}}}$
He (2015)	ReLU and variants	–	$\sigma=\sqrt{\frac{2}{fan_{in}}}$
LeCun (1998)	SELU	–	$\sigma=\sqrt{\frac{1}{fan_{in}}}$



10

BATCH NORMALIZATION (ICML 2015)

- Using weight initialization and non-saturating activation function can significantly reduce the danger of the vanishing/exploding gradients problems at the beginning of training.
- **BUT** it doesn't guarantee that they won't come back during training.
- Why using the **Batch Normalization (BN)** layer?
  - Make sure that the output stays within a fixed range, the same logic that makes us perform the input normalization
  - Ioffe and Szegedy (2015) proposed several advantages of **BN**:
    - **BN** considerably improved all experimental deep learning networks, leading to a huge improvement in the ImageNet classification task.
    - The vanishing gradient problem was strongly reduced to the point that they could use saturating activation functions like tanh and sigmoid.
    - The networks were much less sensitive to the weight initialization.
    - Smoothened the loss landscape which allowed training with larger learning rates, significantly speeding up the learning process
  - **BN** acts like a regularizer, reducing the need for other regularization techniques.

Ioffe and Szegedy, "Batch normalization: accelerating deep network training by reducing internal covariate shift," In Procs. of ICML, 2015

11

BATCH NORMALIZATION (ICML 2015)

- For a hidden layer, it is still arguable whether to insert the **BN** layer before or after the activation function.
  - The original paper of Ioffe and Szegedy (2015) placed the **BN** layer before the activation function.
  - However, some say placing the **BN** layer after the activation function is better.

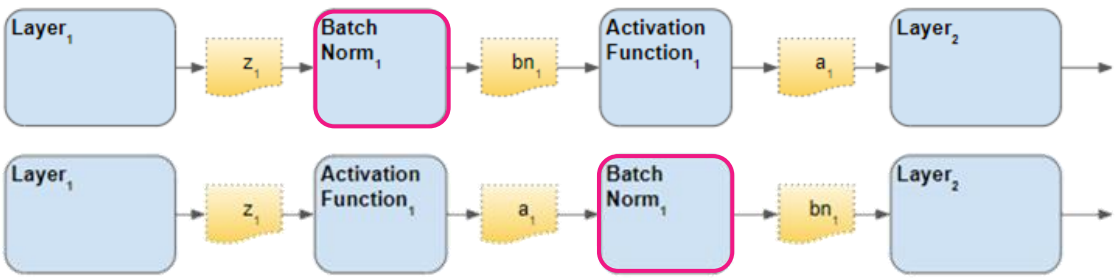


Image credit: (18MAY2021) https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739

12

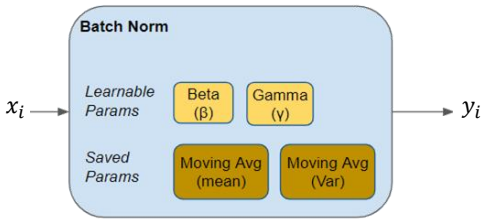
BATCH NORMALIZATION (ICML 2015)

TRAINING

1.	Mini-batch mean	$\mu_B = \frac{1}{\text{batch\_size}} \sum_{i=1}^{\text{batch\_size}} (x_i)$
2.	Mini-batch variance	$(\sigma_B)^2 = \frac{1}{\text{batch\_size}} \sum_{i=1}^{\text{batch\_size}} (x_i - \mu_B)^2$
3.	Normalize	$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{(\sigma_B)^2 + \epsilon}}$
4.	Scale and shift	$y_i = \boldsymbol{\gamma}\hat{x}_i + \boldsymbol{\beta}$
5.	Exponential moving average	$\boldsymbol{\mu_{mov}} = \alpha \cdot \boldsymbol{\mu_{mov}} + (1 - \alpha) * \mu_B$ $(\boldsymbol{\sigma_{mov}})^2 = \alpha \cdot (\boldsymbol{\sigma_{mov}})^2 + (1 - \alpha) * (\sigma_B)^2$

The `momentum` hyperparameter of **BN**

- During training, it simply calculates this EMA but does not do anything with it.
- At the end of training, it simply saves this value as part of the layer's state, for use during the Inference phase.



13

BATCH NORMALIZATION (ICML 2015)

- A single **BN** layer has four parameter vectors:
  - Two learnable (= trainable through backprop) parameter vectors called $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$ that refer to scale and shift (offset) parameter vectors.
  - Two non-learnable (= non-trainable) parameter vectors—Mean Moving Average ($\boldsymbol{\mu_{mov}}$) and Variance Moving Average ($(\boldsymbol{\sigma_{mov}})^2$)—which are saved as part of the state of the **BN** layer.

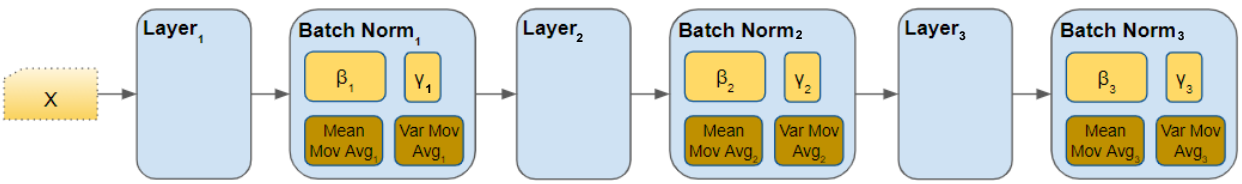


Image credit: (18MAY2021) https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739

14

BATCH NORMALIZATION (ICML 2015)

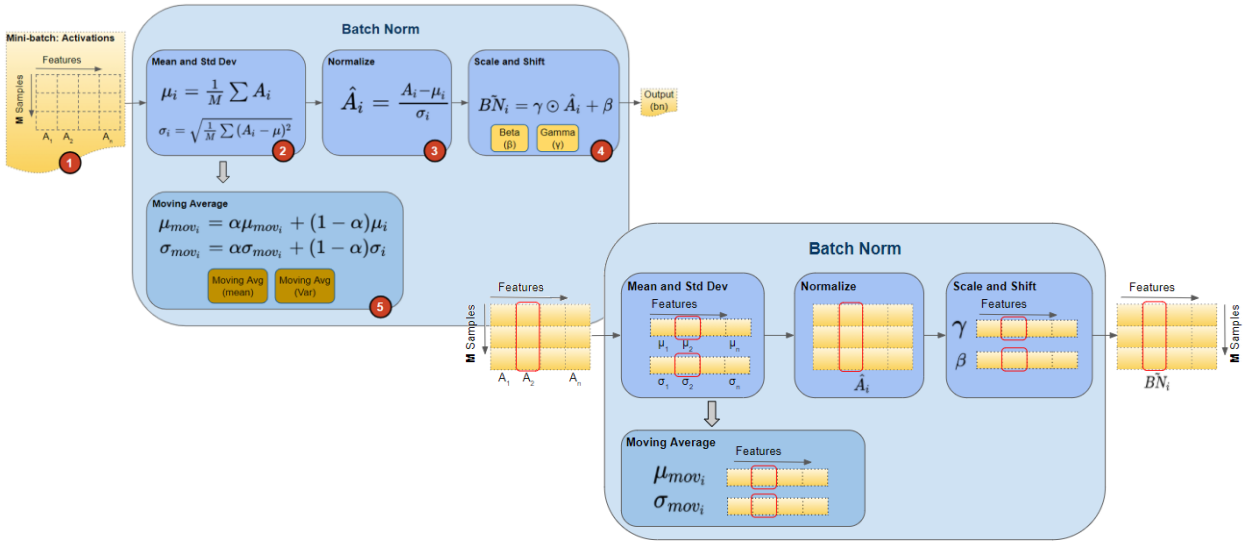


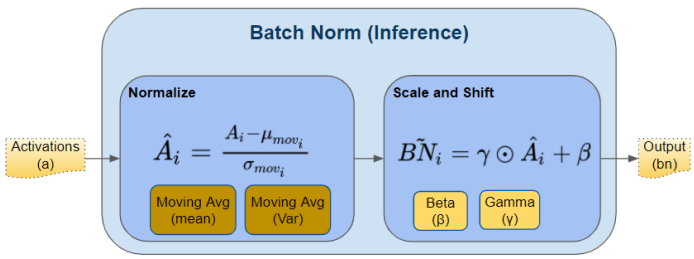
Image credit: (18MAY2021) https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739

15

BATCH NORMALIZATION (ICML 2015)

INFERENCE

- At inference, there may be no batch of samples, or a batch may be too small so that samples may not be independent and identically distributed.
- Most **BN** implementations use the $\boldsymbol{\mu_{mov}}$ and $\boldsymbol{(\sigma_{mov})^2}$ values of the **BN** layer (precomputed during training) to replace the batch input means (μ_B) and SDs (σ_B) during inference.



1.	Mini-batch mean	$\mu_B = \frac{1}{\text{batch\_size}}\sum_{i=1}^{\text{batch\_size}}(x_i)$
2.	Mini-batch variance	$(\sigma_B)^2 = \frac{1}{\text{batch\_size}}\sum_{i=1}^{\text{batch\_size}}(x_i-\mu_B)^2$
3.	Normalize	$\hat{x}_i = \frac{x_i-\mu_B}{\sqrt{(\sigma_B)^2+\epsilon}}$
4.	Scale and shift	$y_i = \boldsymbol{\gamma}\hat{x}_i + \boldsymbol{\beta}$
5.	Exponential moving average	$\boldsymbol{\mu_{mov}} = \alpha \cdot \boldsymbol{\mu_{mov}} + (1-\alpha) * \mu_B$ $(\boldsymbol{\sigma_{mov}})^2 = \alpha \cdot (\boldsymbol{\sigma_{mov}})^2 + (1-\alpha) * (\sigma_B)^2$

Image credit: (18MAY2021) https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739

16

BATCH NORMALIZATION (ICML 2015)

- **BN** has become one of the most-used layers in deep neural networks, to the point that it is often omitted in the diagrams, as it is assumed that **BN** is added after every layer.
- Why not using the **Batch Normalization (BN)** layer? 👎
  - Run time penalty (slower predictions due to the extra computation)
    - This can be solved by fusing the **BN** layer with the previous layer, after training (TFLite's optimizer does this automatically).
  - **BN** adds memory overhead and is a common source of implementation errors.
  - **BN**'s behavior is strongly dependent on the batch size, performing poorly when the per device batch size is too small or too large.
  - **BN** behaves differently during training and inference.
  - Intra-batch information leakage breaks the independence between training samples in the mini-batch, resulting in many implementation errors especially in distributed training.
  - It's difficult to replicate batch normalized models trained on different hardware.
  - **BN** implementation is said to be different on each deep learning framework.
  - The mystery regarding the success of **BN** in deep networks is still unclear.

Google Deepmind, "High-Performance Large-Scale Image Recognition Without Normalization," In arXiv, 2021 https://arxiv.org/abs/2102.06171
Google Deepmind, "Characterizing signal propagation to close the performance gap in unnormalized ResNets," In Procs. of ICLR, 2021 https://arxiv.org/abs/2101.08692

17

BatchNorm1d
BatchNorm2d
BatchNorm3d
LazyBatchNorm1d
LazyBatchNorm2d
LazyBatchNorm3d

CODE REVIEW

BatchNorm1d (Temporal Batch Normalization)

`class torch.nn.BatchNorm1d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True, device=None, dtype=None)` [source]

Applies Batch Normalization over a 2D or 3D input.

Parameters:

- **num_features** (*int*) – number of features or channels C of the input
- **eps** (*float*) – a value added to the denominator for numerical stability. Default: 1e-5
- **momentum** (*float* | *None*) – the value used for the running_mean and running_var computation. Can be set to `None` for cumulative moving average (i.e. simple average). Default: 0.1
- **affine** (*bool*) – a boolean value that when set to `True`, this module has learnable affine parameters. Default: `True`
- **track_running_stats** (*bool*) – a boolean value that when set to `True`, this module tracks the running mean and variance, and when set to `False`, this module does not track such statistics, and initializes statistics buffers `running_mean` and `running_var` as `None`. When these buffers are `None`, this module always uses batch statistics. in both training and eval modes. Default: `True`

Shape:

- Input: (N, C) or (N, C, L), where N is the batch size, C is the number of features or channels, and L is the sequence length
- Output: (N, C) or (N, C, L) (same shape as input)

BatchNorm2d (Spatial Batch Normalization)

`class torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True, device=None, dtype=None)` [source]

Applies Batch Normalization over a 4D input.

4D is a mini-batch of 2D inputs with additional channel dimension. Method described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift .

Parameters:

- **num_features** (*int*) – C from an expected input of size (N, C, H, W)
- **eps** (*float*) – a value added to the denominator for numerical stability. Default: 1e-5
- **momentum** (*float* | *None*) – the value used for the running_mean and running_var computation. Can be set to `None` for cumulative moving average (i.e. simple average). Default: 0.1
- **affine** (*bool*) – a boolean value that when set to `True`, this module has learnable affine parameters. Default: `True`
- **track_running_stats** (*bool*) – a boolean value that when set to `True`, this module tracks the running mean and variance, and when set to `False`, this module does not track such statistics, and initializes statistics buffers `running_mean` and `running_var` as `None`. When these buffers are `None`, this module always uses batch statistics. in both training and eval modes. Default: `True`

Shape:

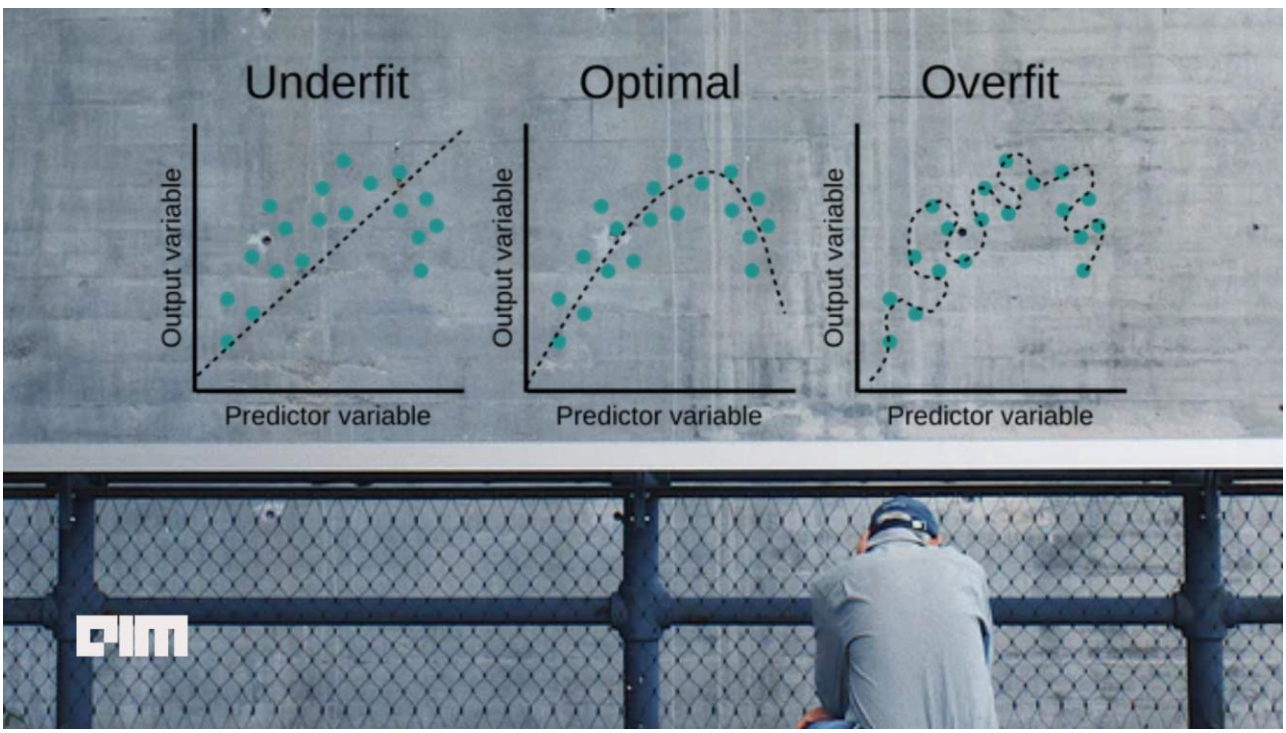
- Input: (N, C, H, W)
- Output: (N, C, H, W) (same shape as input)

18

Problem 2

Overfitting in deep neural networks and how to solve it

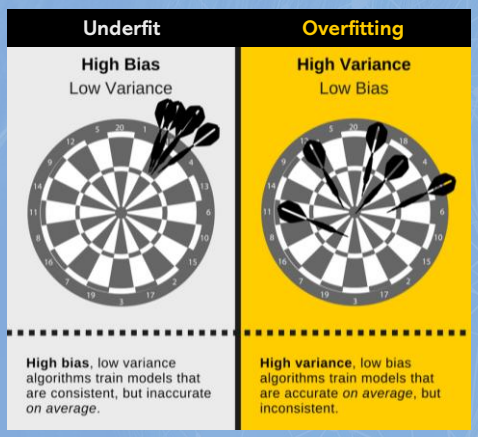
19

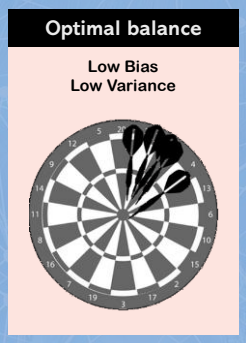


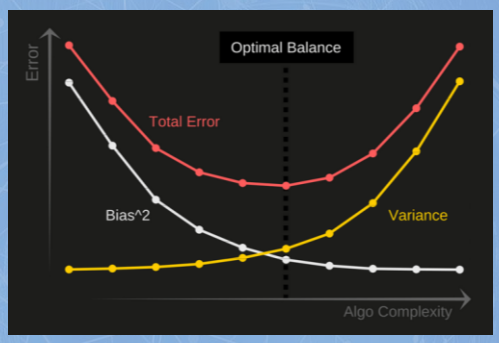
20

Total Error = Bias^2 + Variance + Irreducible Error

Bayes error (a.k.a., irreducible error) is "noise" that can't be reduced by algorithms. It can sometimes be reduced by better data cleaning.







Model suffers from:

1. Only variance (overfitting): val error >> (train error ≅ Bayes error)
2. Only bias (underfitting): (train error ≅ val error) >> Bayes error
3. Both variance (overfitting) and bias (underfitting): val error >> train error >> Bayes error
4. None: train error ≅ val error ≅ Bayes error

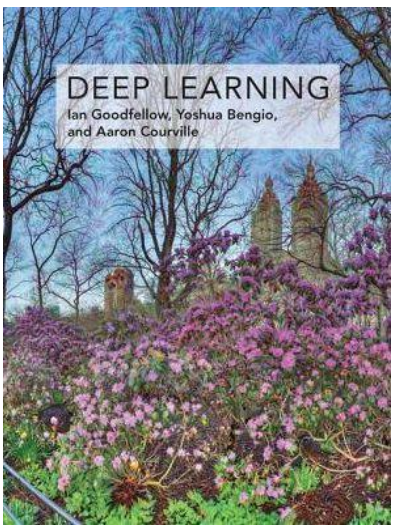
Image credit: https://elitedatascience.com/bias-variance-tradeoff

21

REGULARIZATION

"Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error."

Reduce overfit



22

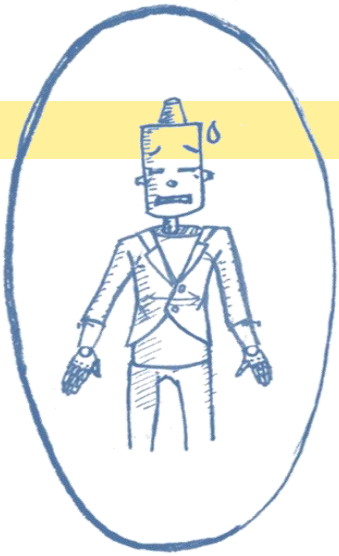
Regularization in Deep NN

- ~~Batch Normalization (2015)~~
- Early stopping
- L1 and L2 regularization
- Dropout (2012)

Lower the complexity of the network during training

- Data augmentation
- …

Increase the training dataset



23

Early Stopping

- **Double descent phenomenon** occurs in CNNs, ResNets, and Transformers.
  - Performance first improves, then gets worse, and then improves again with increasing model size, data size, or training time. ⇒ This implies that, finally, big models will converge. As long as we, in practice, have enough computational resources to train them.
  - This effect is often avoided through careful regularization. While this behavior appears to be fairly universal, we don't yet fully understand why it happens, and view further study of this phenomenon as an important research direction.

Early Stopping in
- Manual implementation for learning/research to maintain full control.
- Use the `EarlyStopping` callback from PyTorch Lightning

[5DEC2019] https://openai.com/blog/deep-double-descent/
[10DEC2018] https://machinelearningmastery.com/how-to-stop-training-deep-neural-networks-at-the-right-time-using-early-stopping/

24

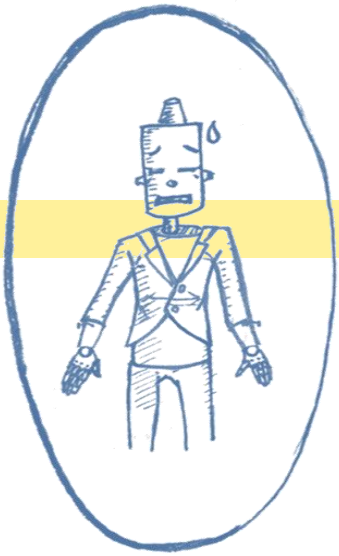
Regularization in Deep NN

- ~~Batch Normalization (2015)~~
- Early stopping
- L1 and L2 regularization
- Dropout (2012)

Lower the complexity of the network during training

- Data augmentation
- …

Increase the training dataset



25

L1 & L2 REGULARIZATION

- **L1** and **L2** are the most common types of regularization that update the general cost/loss function by adding another term known as the regularization term (a.k.a., the penalty term).

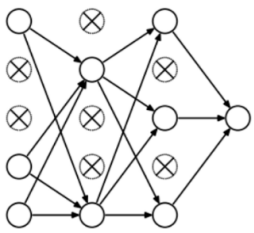
A loss function (e.g., BCE loss) | **The penalty term**

$$-\sum_{i=1}^{\text{output size}}[y_i \cdot \log \hat{y}_i + (1-y_i) \cdot \log(1-\hat{y}_i)] + \text{RegularizationTerm}$$

- What are the consequences of having higher loss?

Term to be optimized

LOSS ⬆ w_i ⬇



26

L1 & L2 RE

L2 Regularization and Weight Decay are only the same thing for vanilla SGD, but as soon as we add momentum, or use a more sophisticated optimizer like Adam, L2 regularization and Weight Decay become different.

- **L2 Regularization** (a.k.a., Weight decay)
  - The most common type of all regularization techniques
  - Encourage the weight values towards zero (but not exactly zero)
  - The regularization term is defined as the Euclidean Norm (or L2 norm) of the weight matrices.
- **L1 Regularization** (a.k.a., Lasso regression)
  - Encourage the weight values to be zero, resulting in a sparse model
  - The regularization term is the sum of the absolute values of the weight parameters in a weight matrix.
- λ is sometimes called as the regularization rate/parameter and is an additional hyperparameter to determine how much we regularize our model.

$$Loss = Error(y, \hat{y})$$

Loss function with no regularisation

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} |w_i|$$

Loss function with L1 regularisation

$$Loss = Error(y, \hat{y}) + \lambda \sum_{i=1}^{N} w_i^2$$

Loss function with L2 regularisation

27

CODE REVIEW

```
# --- 1. The Standard "Weight Decay" (L2 Regularization) ---
# In PyTorch, L2 is almost always handled by the Optimizer.
# This is equivalent to adding L2 regularization to every layer in Keras.
model = torch.nn.Sequential(torch.nn.Linear(10, 5), torch.nn.ReLU(), torch.nn.Linear(5, 2))

# 'weight_decay' is the L2 penalty coefficient (lambda)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-5)
```

```
# --- 2. The Manual Way (L1 Regularization) ---
# PyTorch does not have a built-in 'l1_decay' parameter in optimizers.
# Because L1 is non-differentiable at zero, it is manually added to the loss.
def train_step_l1(model, data, target, optimizer, l1_lambda=1e-5):
    optimizer.zero_grad()
    output = model(data)
    criterion_loss = nn.CrossEntropyLoss()(output, target)

    # Calculate L1 penalty manually
    l1_penalty = sum(p.abs().sum() for p in model.parameters())

    # Total loss
    loss = criterion_loss + (l1_lambda * l1_penalty)

    loss.backward()
    optimizer.step()
```

28

CODE REVIEW

```python
# --- 3. Layer-Specific Regularization (The "Keras" Way) ---
# To regularize specific layers (e.g., just the weights of the first layer but not the bias),
# use Parameter Groups.
classifier = torch.nn.Sequential( torch.nn.Linear(100, 50), torch.nn.ReLU(), torch.nn.Linear(50, 10) )


# Separate parameters into groups with different settings
optimizer = torch.optim.Adam([
    {'params': classifier[0].weight, 'weight_decay': 0.01}, # Strong L2
    {'params': classifier[2].weight, 'weight_decay': 0.0},  # No L2
    {'params': [classifier[0].bias, classifier[2].bias], 'weight_decay': 0.0} # No L2 on bias
], lr=1e-3)
```

```python
# --- 4. Decoupled Weight Decay Regularization (AdamW) ---
# When using AdamW, be careful with the default value of the 'weight_decay' parameter.
# It MUST be set to any non-zero value (e.g., weight_decay=1e-2) to enable the regularization.


# AdamW differs from standard Adam because it applies the decay directly
# to the weights rather than incorporating it into the gradient calculation.
# This "decoupling" prevents the adaptive learning rate from
# inadvertently shrinking the regularization effect over time.

optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3, weight_decay=0.01)
```


OTHER REGULARIZATION

```python
keras.layers.Dense(
    units,
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    lora_rank=None,
    lora_alpha=None,
    **kwargs
)
```

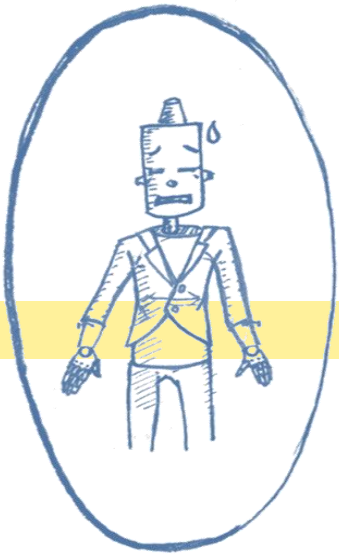
- `kernel_regularizer`: Apply regularization only to the weights of the layer
  - Penalize large weights. Try to reduce weights (excluding bias).
- `bias_regularizer`: Apply regularization only to the biases of the layer
  - Penalize large biases. Try to reduce bias.
  - However, regularizing bias values may not change the results significantly.
- `activity_regularizer`: Apply regularization only to the output of the layer
  - Penalize large outputs. Thus, it will reduce the weights and adjust the bias so the layer's output is the smallest.
  - A use case to get sparse outputs/representations from AutoEncoder https://stackoverflow.com/questions/44495698/keras-difference-between-kernel-and-activity-regularizers

Regularization in Deep NN

- ~~Batch Normalization (2015)~~
- Early stopping
- L1 and L2 regularization
- Dropout (2012)
- Data augmentation
- …

Lower the complexity of the network during training

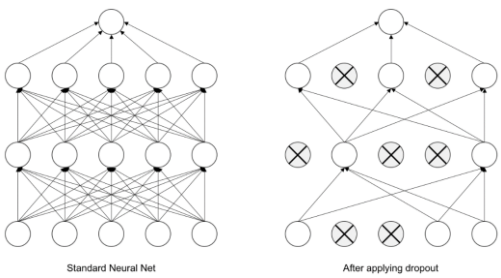
Increase the training dataset



31

DROPOUT (2012)

- **Dropout** by Hinton et al. (2012) is one of the most popular regularization techniques for deep neural networks.
- **During training:**
  - At every training step, every neuron (including the input neurons, but always excluding the output neurons) has a probability p of being temporarily "dropped out."
  - p is a new hyperparameter called the dropout rate, and typically set between 10% to 50% (20-30% in RNN and 40-50% in CNN).
- Why does **dropout** help reduce overfit?
  - At each training step, a smaller network is trained.
  - A network can't rely on any one feature (= can't bet a large value on any one weight) and has to spread out weights, resulting in weight shrinking (similar to the L2 regularization).



Hinton et al., "Improving neural networks by preventing co-adaptation of feature detectors," CoRR, 2012 https://arxiv.org/abs/1207.0580

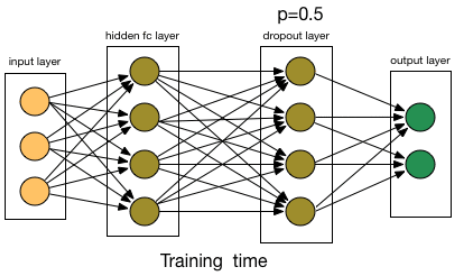
32

DROPOUT (2012)

- **Dropout** also helps explore many sub-networks and can be thought of as an ensemble technique. Ensemble models usually perform better than a single model as they capture more randomness.

- Where to apply **dropout**?
  - Dropout can be applied to any layer but it is uncommon to apply dropout to input (and output) layers.
  - We may apply dropout with a high dropout rate to a layer with a large number of trainable parameters (as it has a high risk of overfitting), and a low dropout rate otherwise.
  - In the practice of computer vision tasks, we usually apply dropout only to the neurons in the top one to three (FC) layers (excluding the output layer).

- **During inference:** To make sure that the sum over all inputs is unchanged during training (when **dropout** is active) and during inference (when **dropout** is inactive):
  - During inference, multiply each input connection weight (of the layer after dropout) by the keep probability $(1-p)$, **OR**
  - During training, divide each neuron's output (of the layer before dropout) by the keep probability



33

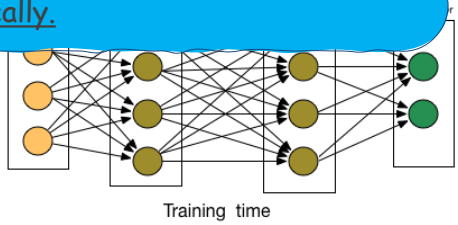
DROPOUT (2012)

- **Dropout** also helps explore many sub-networks and can be thought of as an ensemble technique. Ensemble models usually perform better than a single model as they capture more randomness.

Because Dropout behaves differently during training (`model.train()`) and evaluation (`model.eval()`), a direct comparison between training loss and validation loss can be misleading.

- A model may be overfitting the training set and yet have similar training and validation losses.
- To get a true diagnostic of overfitting, evaluating the training loss in '`model.eval()`' mode (without Dropout) periodically.

- **During inference:** To make sure that the sum over all inputs is unchanged during training (when **dropout** is active) and during inference (when **dropout** is inactive):
  - During inference, multiply each input connection weight (of the layer after dropout) by the keep probability $(1-p)$, **OR**
  - During training, divide each neuron's output (of the layer before dropout) by the keep probability



34

CODE REVIEW

- Dropout
- Dropout1d
- Dropout2d
- Dropout3d
- AlphaDropout
- FeatureAlphaDropout

Dropout

class torch.nn.Dropout(*p=0.5, inplace=False*) [source]

During training, randomly zeroes some of the elements of the input tensor with probability `p`.

The zeroed elements are chosen independently for each forward call and are sampled from a Bernoulli distribution.

Each channel will be zeroed out independently on every forward call.

This has proven to be an effective technique for regularization and preventing the co-adaptation of neurons as described in the paper Improving neural networks by preventing co-adaptation of feature detectors .

Furthermore, the outputs are scaled by a factor of $\frac{1}{1-p}$ during training. This means that during evaluation the module simply computes an identity function.

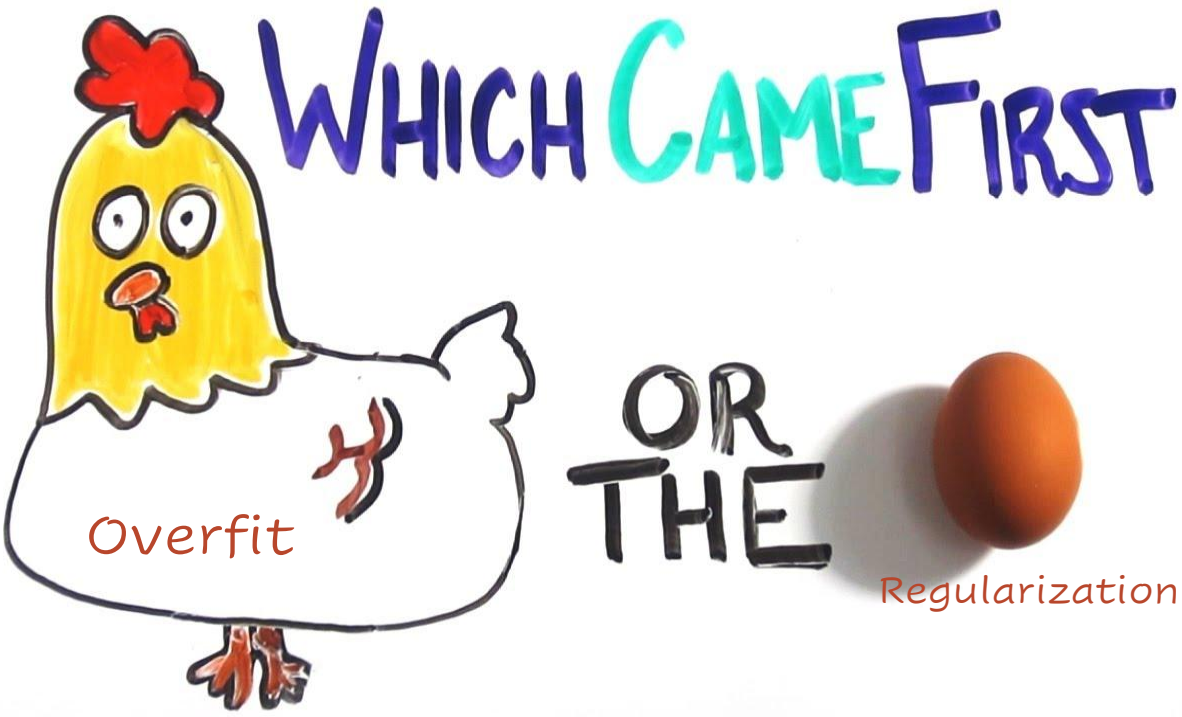
Parameters:

- **p** (*float*) – probability of an element to be zeroed. Default: 0.5
- **inplace** (*bool*) – If set to `True`, will do this operation in-place. Default: `False`

Shape:

- Input: $(*)$. Input can be of any shape
- Output: $(*)$. Output is of the same shape as input

35



36

37

Coding

Combine everything and implement a program with PyTorch

38

EX 1: MLP for Tabular Data

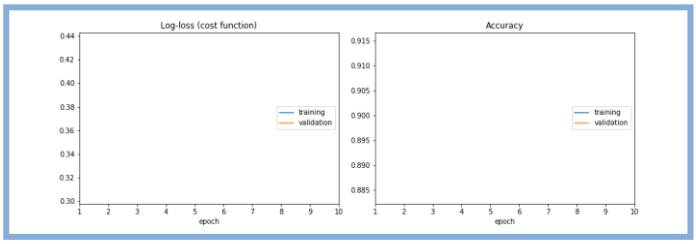
- Boston Housing Price dataset:
  - https://lib.stat.cmu.edu/datasets/boston (required many CSV preprocess)

- **Tips & Tricks:**
  - In practice, K-fold cross-validation is not frequently used in deep learning due to the very expensive cost of training K different deep learning models (This issue was mentioned in the famous YouTube class of Stanford @34:00 https://www.youtube.com/watch?v=OoUX-nOEjG0&list=PL3FW7Lu3i5JvHM8ljYj-zLfQRF3EO8sYv&index=2).
  - The model obtained at the last epoch may not be the best model that yields the minimum validation loss.
  - For long (and serious) training, try using interactive visualization tools to log and visualize training results (e.g., train loss, val loss, image) in real time.

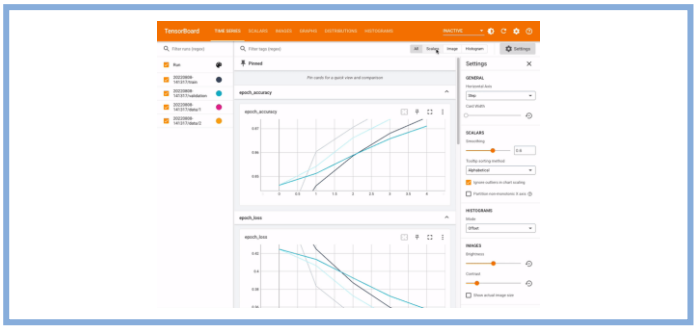
39

Interactive Visualization

- **Livelossplot:**
  https://github.com/stared/livelossplot
  - Simple installation and usage
  - Jupyter notebook compatibility



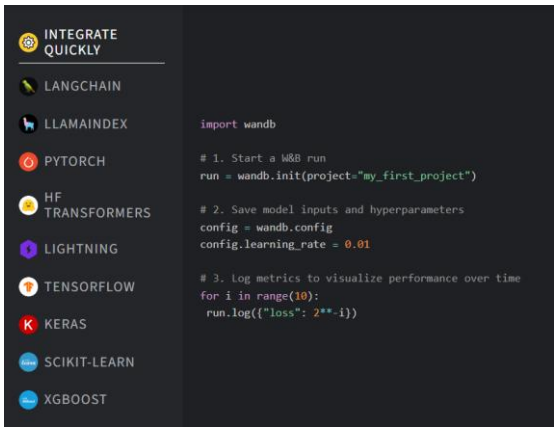
- **TensorBoard:**
  - Popular visualization and logging tools in deep learning for both TensorFlow and PyTorch users
  - Can log many types of information
  - Open source, run locally



40

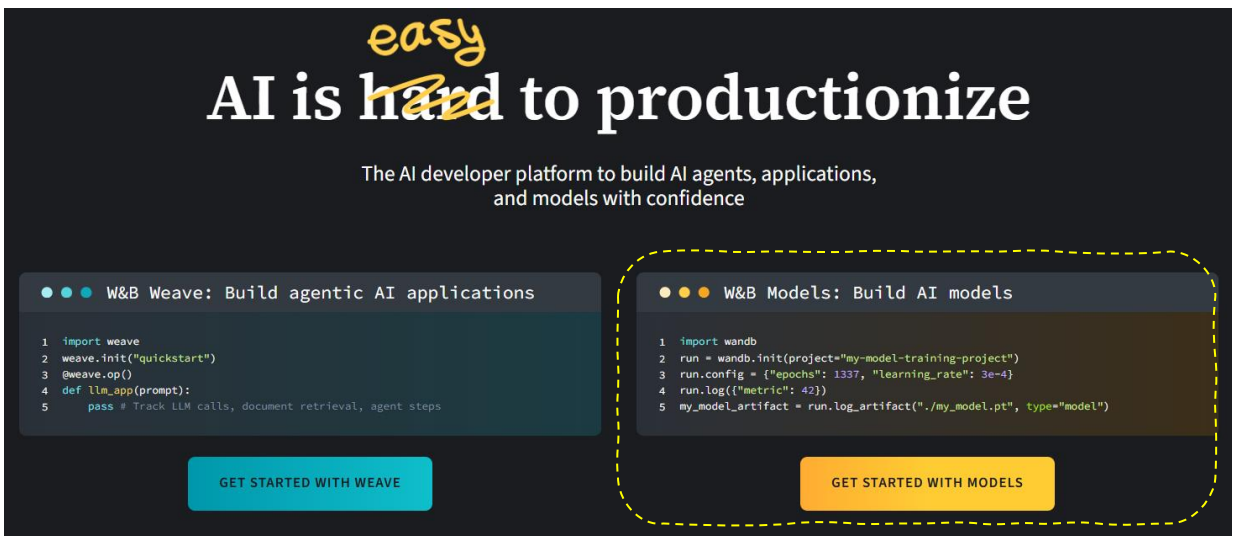
Interactive Visualization

- **Weights & Biases (WandB):** https://wandb.ai/site
  - Compared with TensorBoard:
    - TensorBoard is an open-source tool that runs locally so it's difficult to track everything when working on different machines.
    - WandB offers managed services that can be deployed on-premises but also run in the cloud.
    - WandB provides wider functionality than TensorBoard, covering experiment tracking, dataset versioning, and model management.
    - WandB has a lot of features that enable team collaboration, something that's missing in TensorBoard.
    - Compared to WandB, TensorBoard doesn't scale well with more experiments.
    - More in https://neptune.ai/vs/wandb-tensorboard



41

Interactive Visualization



https://wandb.ai/site/

42

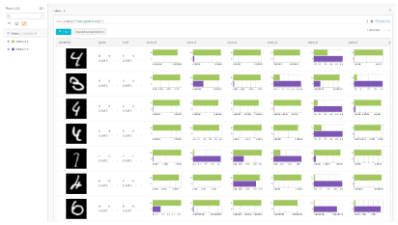
Interactive Visualization

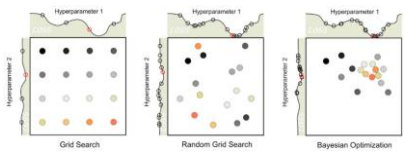
W&B

43

Interactive Visualization

- **Weights & Biases (WandB):** https://wandb.ai/site
  - Some interesting features:
    - Pandas-like summarization table that allows easy filtering
    - W&B Artifact for data and model versioning
    - W&B Table for log, visualize, and dynamically explore data
      - https://wandb.ai/stacey/mnist-viz/reports/Guide-to-W-B-Tables--Vmlldzo2NTAzOTk
      - https://docs.wandb.ai/models/tutorials/tables
    - W&B Sweep for hyperparameter tuning
      - https://docs.wandb.ai/models/tutorials/sweeps
      - https://docs.wandb.ai/models/sweeps/sweep-config-keys
      - https://docs.wandb.ai/models/sweeps/walkthrough
      - https://docs.wandb.ai/models/sweeps
      - **Kaggle:** https://www.kaggle.com/code/samuelcortinhas/weights-biases-tutorial-beginner, https://www.kaggle.com/code/samuelcortinhas/advanced-wandb-hyper-parameter-tuning-sweeps
    - Visualizing W&B Sweep results includes a hyperparameter importance plot that helps us find out which hyperparameters actually matter.
      - https://docs.wandb.ai/models/sweeps/visualize-sweep-results
        https://docs.wandb.ai/models/app/features/panels/parameter-importance
      - https://wandb.ai/wandb_fc/articles/reports/Find-The-Most-Important-Hyperparameters-In-Seconds--Vmlldzo1NDMxNTQz



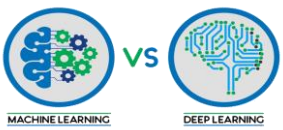


44

Homework

45

Structured Data: Traditional



- **Objective:** Which one is better for structured data, traditional ML or MLP?
- Compare a self-designed MLP network with one or more traditional machine learning techniques
  - Choose at least one tabular dataset to share among group members; no duplication with those already used in the class; no duplication between groups; PyTorch's built-in datasets are not allowed.
  - Experiment with the hyperparameter tuning by your own self, comparing several combinations of hyperparameters and finding the best one
  - Conclude implementation detail, comparative results (both technical and design aspects), and recommendation about using MLP vs. using traditional ML for structure data

46

Tabular Deep Learning

Revisiting Deep Learning Models for Tabular Data

Yury Gorishniy*†‡ Ivan Rubachev†♣ Valentin Khrulkov† Artem Babenko†♣

† Yandex, Russia
‡ Moscow Institute of Physics and Technology, Russia
♣ National Research University Higher School of Economics, Russia

Abstract

The existing literature on deep learning for tabular data proposes a wide range of novel architectures and reports competitive results on various datasets. However, the proposed models are usually not properly compared to each other and existing works often use different benchmarks and experiment protocols. As a result, it is unclear for both researchers and practitioners what models perform best. Additionally, the field still lacks effective baselines, that is, the easy-to-use models that provide competitive performance across different problems.

In this work, we perform an overview of the main families of DL architectures for tabular data and raise the bar of baselines in tabular DL by identifying two simple and powerful deep architectures. The first one is a ResNet-like architecture which turns out to be a strong baseline that is often missing in prior works. The second model is our simple adaptation of the Transformer architecture for tabular data, which outperforms other solutions on most tasks. Both models are compared to many existing architectures on a diverse set of tasks under the same training and tuning protocols. We also compare the best DL models with Gradient Boosted Decision Trees and conclude that there is still no universally superior solution. The source code is available at https://github.com/yandex-research/rtdl.

Figure 1: The FT-Transformer architecture. Firstly, Feature Tokenizer transforms features to embeddings. The embeddings are then processed by the Transformer module and the final representation of the [CLS] token is used for prediction.

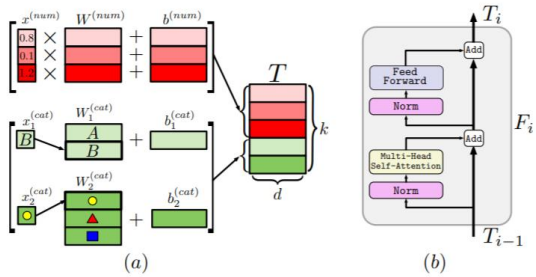
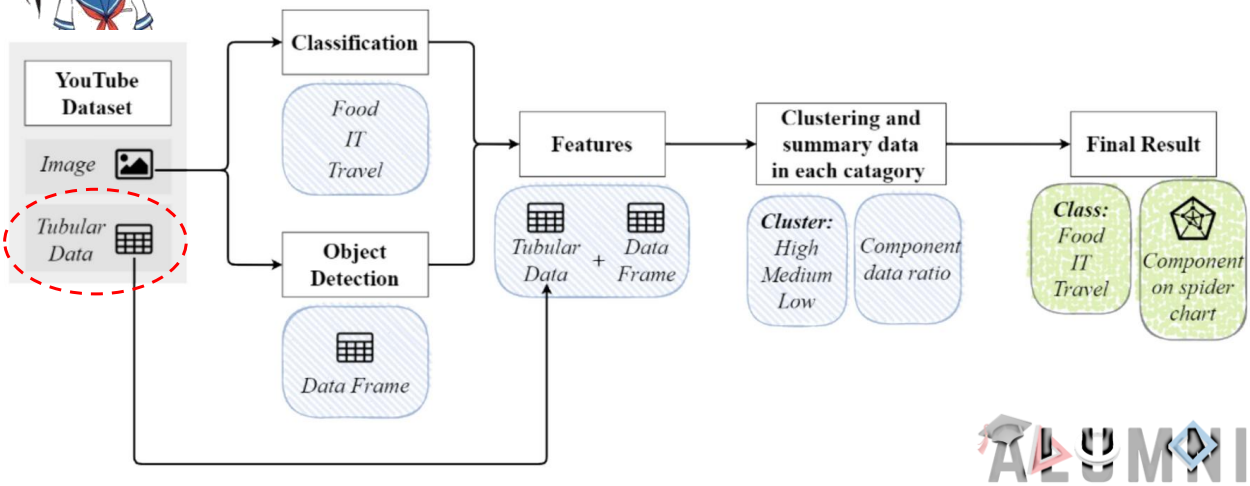


Figure 2: (a) Feature Tokenizer; in the example, there are three numerical and two categorical features; (b) One Transformer layer.

"Revisiting Deep Learning Models for Tabular Data", In Procs. of NeurIPS, 2021 https://arxiv.org/abs/2106.11959

47

YouTube thumbnail design



"YouTube thumbnail design recommendation systems using image-tabular multimodal data for Thai's YouTube thumbnail," Social Network Analysis and Mining, 2024. https://link.springer.com/article/10.1007/s13278-024-01317-7

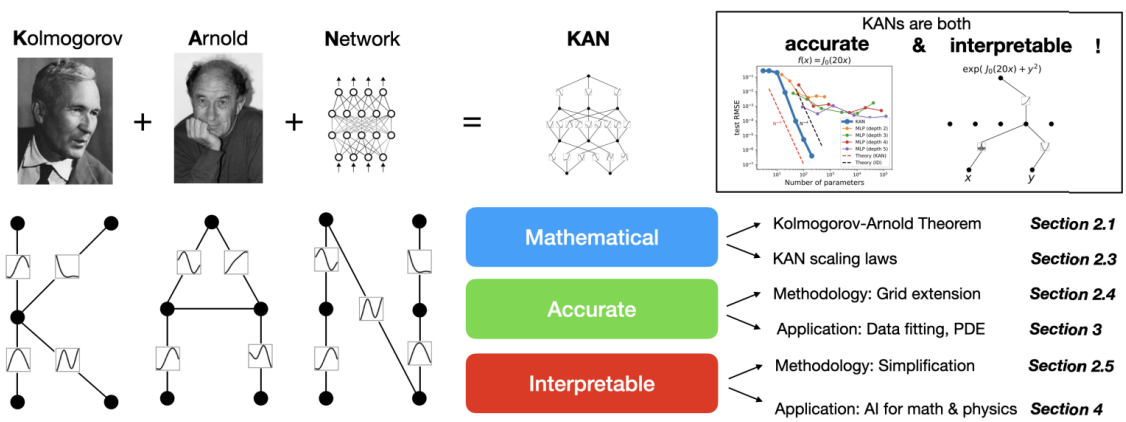
48

AI ETC.

49

Kolmogorov-Arnold Network (arxiv 2024)

- **KANs** are promising alternatives to MLPs.
- **KANs** have strong mathematical foundations just like MLPs.



"KAN: Kolmogorov-Arnold Networks," arxiv30APR2024 https://arxiv.org/abs/2404.19756 , https://github.com/KindXiaoming/pykan , https://kindxiaoming.github.io/pykan/

50

Kolmogorov-Arnold Representation Theorem

Kolmogorov-Arnold Representation Theorem (Superposition theorem) established that if f is a multivariate continuous function, then f can be written as a finite composition of continuous functions of a single variable and the binary operation of addition:

$$f(x_1, x_2, \ldots, x_n) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right)$$

In other words, this theorem states that any continuous function of n variables can be represented as a composition of $2n+1$ univariate functions.

51

Kolmogorov-Arnold Representation Theorem

# Bedrooms	Area (Sq Ft)	Age	Price
2	18	25	**1000**
1	6	4	**500**
2	12	35	**1200**

Instead of writing in the "multi-variate" form:

Price = F(*Bedrooms, Area, Age*)

We can write:

Price = $f(\phi_B(Bedrooms) + \phi_{Sq}(Area) + \phi_A(Age))$

Or more generally:

Price = $f_1(\phi_{B1}(Bedrooms) + \phi_{Sq1}(Area) + \phi_{A1}(Age)) +$
$f_2(\phi_{B2}(Bedrooms) + \phi_{Sq2}(Area) + \phi_{A2}(Age)) +$
$f_3(\phi_{B3}(Bedrooms) + \phi_{Sq3}(Area) + \phi_{A3}(Age)) +$
....
$f_m(\phi_{B\,m}(Bedrooms) + \phi_{Sq\,m}(Area) + \phi_{A\,m}(Age))$

Hope this formula makes more sense now!

$$Y = \sum_{p=1}^{m} f_p \left(\sum_{q=1}^{n} \phi_{pq}(X_q) \right)$$

Image source: 11MAY2024 https://www.youtube.com/watch?v=7zpz_AIFW2w

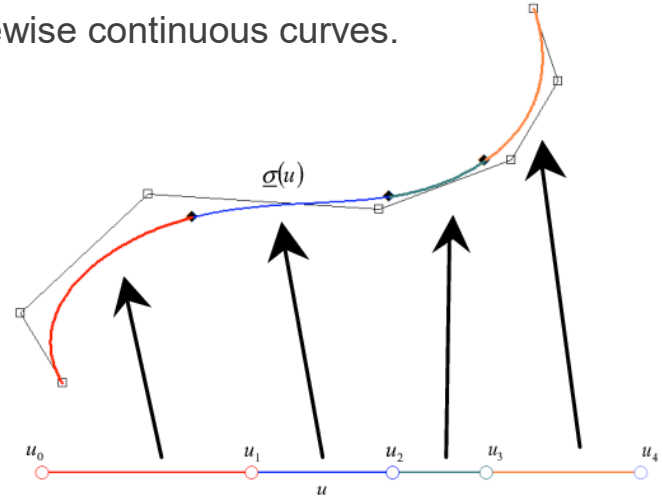
52

B-splines

B-splines are the union of piecewise continuous curves.

Example of piecewise equations for a cubic B-spline function

$$b^3 : \mathbb{R} \to \mathbb{R}\,, \qquad x \mapsto \begin{cases} 0 & \text{für } x \le -2 \\ \frac{1}{6}x^3 + x^2 + 2x + \frac{4}{3} & \text{für } -2 < x \le -1 \\ -\frac{1}{2}x^3 - x^2 + \frac{2}{3} & \text{für } -1 < x \le 0 \\ \frac{1}{2}x^3 - x^2 + \frac{2}{3} & \text{für } 0 < x \le 1 \\ -\frac{1}{6}x^3 + x^2 - 2x + \frac{4}{3} & \text{für } 1 < x \le 2 \\ 0 & \text{für } x > 2 \end{cases}$$



53

Kolmogorov-Arnold Network (arxiv 2024)

Model	Multi-Layer Perceptron (MLP)	Kolmogorov-Arnold Network (KAN)
Theorem	Universal Approximation Theorem	Kolmogorov-Arnold Representation Theorem
Formula (Shallow)	$f(\mathbf{x}) \approx \sum_{i=1}^{N(\epsilon)} a_i \sigma(\mathbf{w}_i \cdot \mathbf{x} + b_i)$	$f(\mathbf{x}) = \sum_{q=1}^{2n+1} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right)$
Model (Shallow)	(a) *fixed* activation functions on *nodes*; *learnable* weights on *edges*	(b) *learnable* activation functions on *edges*; sum operation on *nodes*
Formula (Deep)	$\mathrm{MLP}(\mathbf{x}) = (\mathbf{W}_3 \circ \sigma_2 \circ \mathbf{W}_2 \circ \sigma_1 \circ \mathbf{W}_1)(\mathbf{x})$	$\mathrm{KAN}(\mathbf{x}) = (\mathbf{\Phi}_3 \circ \mathbf{\Phi}_2 \circ \mathbf{\Phi}_1)(\mathbf{x})$
Model (Deep)	(c) $\mathbf{W}_3$, σ_2, $\mathbf{W}_2$, σ_1, $\mathbf{W}_1$; *nonlinear, fixed*; *linear, learnable*	(d) $\mathbf{\Phi}_3$, $\mathbf{\Phi}_2$, $\mathbf{\Phi}_1$; *nonlinear, learnable*

"KAN: Kolmogorov-Arnold Networks," arxiv 30APR2024 https://arxiv.org/abs/2404.19756 , https://github.com/KindXiaoming/pykan , https://kindxiaoming.github.io/pykan/

54

Kolmogorov-Arnold Network (arxiv 2024)

- **KANs** use learnable activation functions on edges, entirely eliminating linear weight matrices and replacing them with learnable 1D spline functions. The nodes in a **KAN** simply sum incoming signals without applying any non-linearities.
  - Generally, splines excel in accurately representing low-dimensional functions but struggle with high-dimensional data due to the curse of dimensionality.
  - On the other hand, MLPs are proficient in feature learning but may falter in optimizing univariate functions.
  - Hence, **KANs** merge the accuracy of splines with the feature learning capability of MLPs.

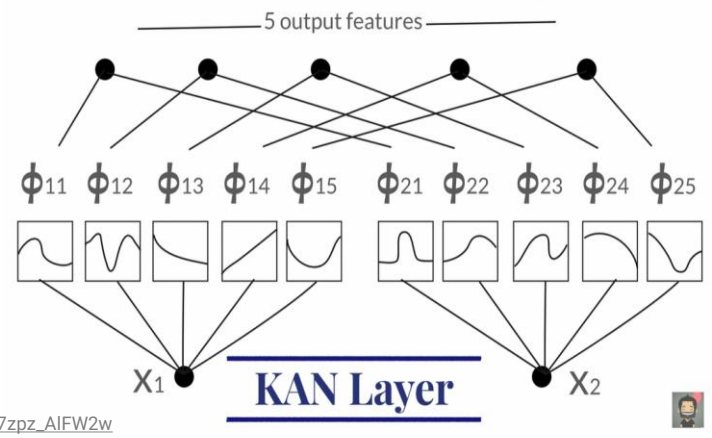
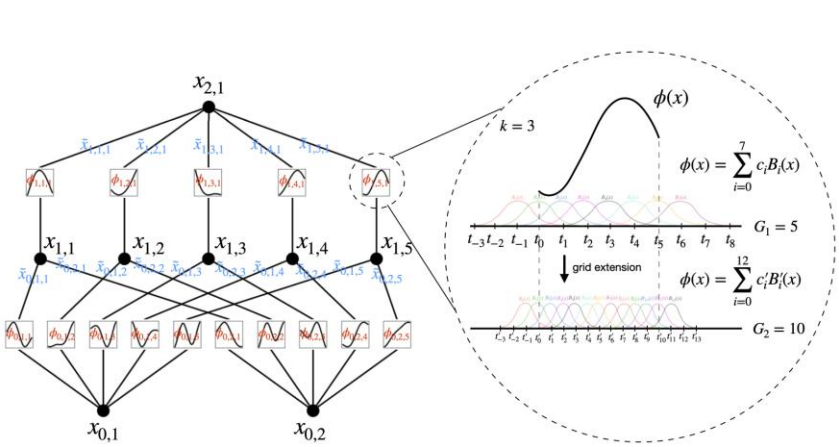
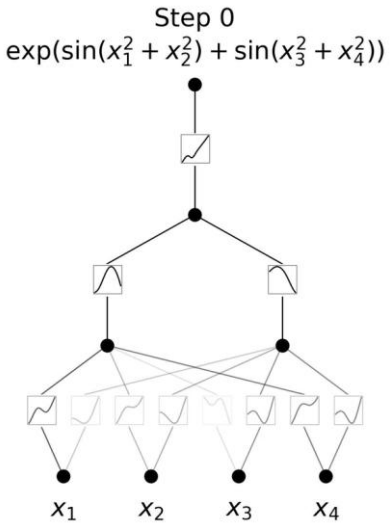


Image source: 11MAY2024 https://www.youtube.com/watch?v=7zpz_AlFW2w

55

Kolmogorov-Arnold Network (arxiv 2024)

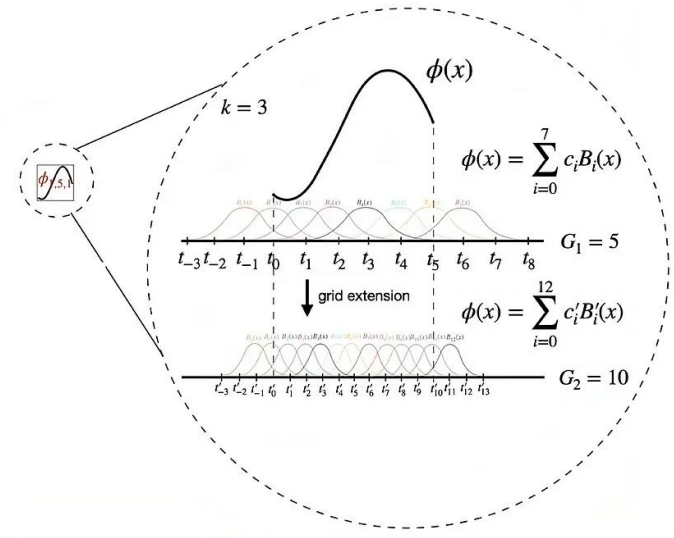




"KAN: Kolmogorov-Arnold Networks," arxiv 30APR2024 https://arxiv.org/abs/2404.19756 , https://github.com/KindXiaoming/pykan , https://kindxiaoming.github.io/pykan/

56

Kolmogorov-Arnold Network (arxiv 2024)



The general formula for a spline in the context of **KANs** can be expressed using B-splines as:

$$\text{spline}(x) = \sum_i c_i B_i(x)$$

- $\text{spline}(x)$ represents the spline function.
- c_i are the coefficients that are optimized during training.
- $B_i(x)$ are the B-spline basis functions defined over a grid.
- The grid points define the intervals where each basis function B_i is active and significantly affects the shape and smoothness of the spline.

"KAN: Kolmogorov-Arnold Networks," arxiv 30APR2024 https://arxiv.org/abs/2404.19756 , https://github.com/KindXiaoming/pykan , https://kindxiaoming.github.io/pykan/

57

Kolmogorov-Arnold Network (arxiv 2024)

- Pros of **KANs** (according to the **KAN** paper):
  - Interpretability
    - Because learnable results are functions not just a weight matrix for linear equations.
  - Continual learning without catastrophic forgetting
    - MLPs usually use global activations and any local change may propagate uncontrollably to regions far away, destroying the information being stored there.
    - In **KANs**, since spline bases are local, a sample will only affect a few nearby spline coefficients, leaving far-away coefficients intact.
  - Parameter efficient
  - Beating the curse of dimensionality
  - Additional complexity without retraining
    - One can first train a **KAN** with fewer parameters and then extend it to a **KAN** with more parameters by simply making its spline grids finer, without the need to retrain the larger model from scratch.

- Cons of **KANs** (according to the **KAN** paper):
  - Slow training, usually 10x slower than MLPs at the same numbers of params
  - Not GPU-efficient as different activation functions cannot leverage batch computation
  - Lack of testing in real-world ML datasets (e.g., no test for sequential data and large-scale dataset)
  - Potential overfit, especially in scenarios with limited data

Recommended materials:
- Easy overview: (11MAY2024): https://www.youtube.com/watch?v=7zpz_AIFW2w
- More in detail (11MAY2024): https://www.youtube.com/watch?v=-PFIkkwWdnM

58

Kolmogorov-Arnold Network (arxiv 2024)

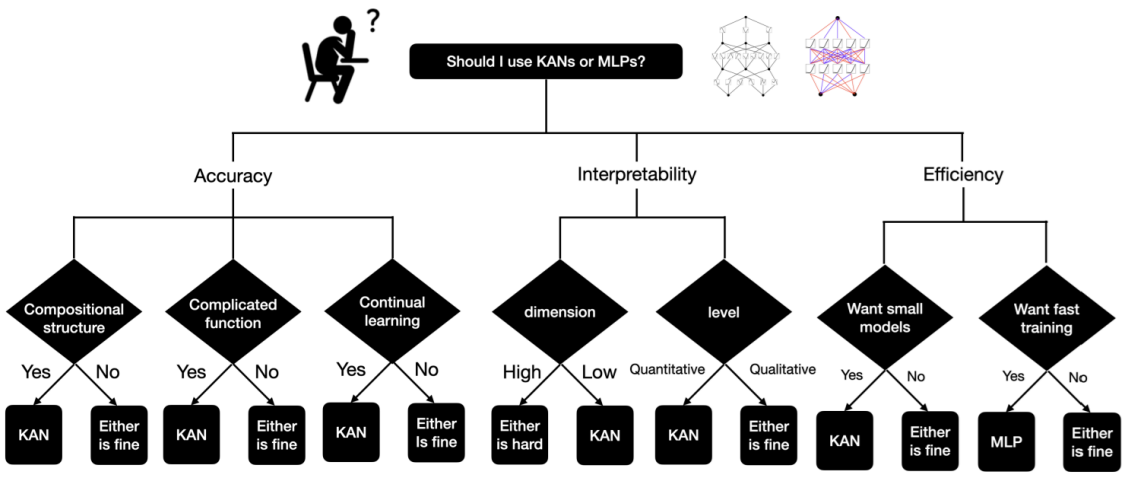


Figure 6.1: Should I use KANs or MLPs?

"KAN: Kolmogorov-Arnold Networks," arxiv 30APR2024 https://arxiv.org/abs/2404.19756 , https://github.com/KindXiaoming/pykan , https://kindxiaoming.github.io/pykan/

59

60