Course series: Deep Learning for NLP

Practical Considerations

Lecture # 4b

Hassan Sajjad and Fahim Dalvi
Qatar Computing Research Institute, HBKU

Recap

So far, we have seen:

- Feed forward neural networks
- Neural network language model
- Recurrent neural networks
- Sequence to sequence models with attention mechanism

In this Lecture

Practical considerations

Data Preprocessing

- Batch processing
- Padding
- Class imbalance
- Data normalization

Network Setup

- Initialization
- Dropouts
- Ensembles
- Residual Connections
- Batch norm

Miscellaneous Tweaks and Tips

- Gradient Accumulation
- Quantization
- LoRa
- Flash Attention
- vLLM

Batch Processing

Practical Considerations

Batch Processing

In the previous implementations, we've computed the objective function and gradient for every instance, adjusted our parameters and then continued with the next instance

Batch Processing

In the previous implementations, we've computed the objective function and gradient for every instance, adjusted our parameters and then continued with the next instance

This approach is known as ***stochastic gradient descent*** (SGD)

Batch Processing

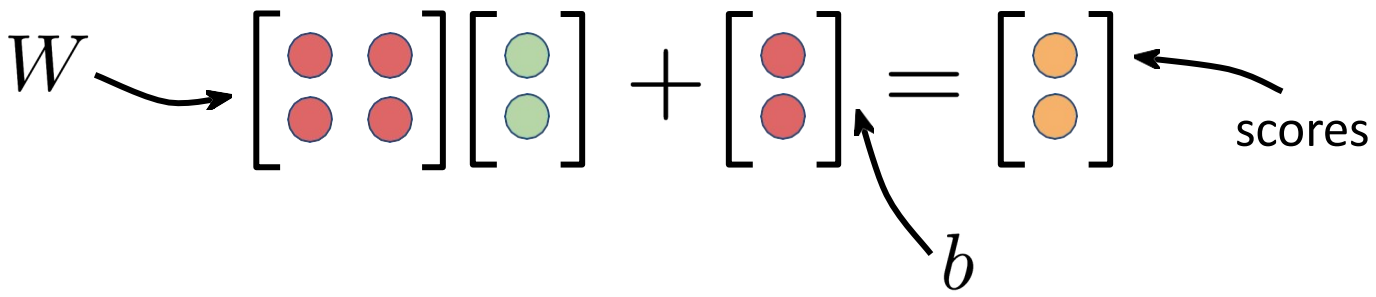
Another approach is to accumulate the gradients over all instances, and then do a single update to the parameters - this approach is known as ***Batch gradient descent***

Batch Processing

Another approach is to accumulate the gradients over all instances, and then do a single update to the parameters - this approach is known as ***Batch gradient descent***

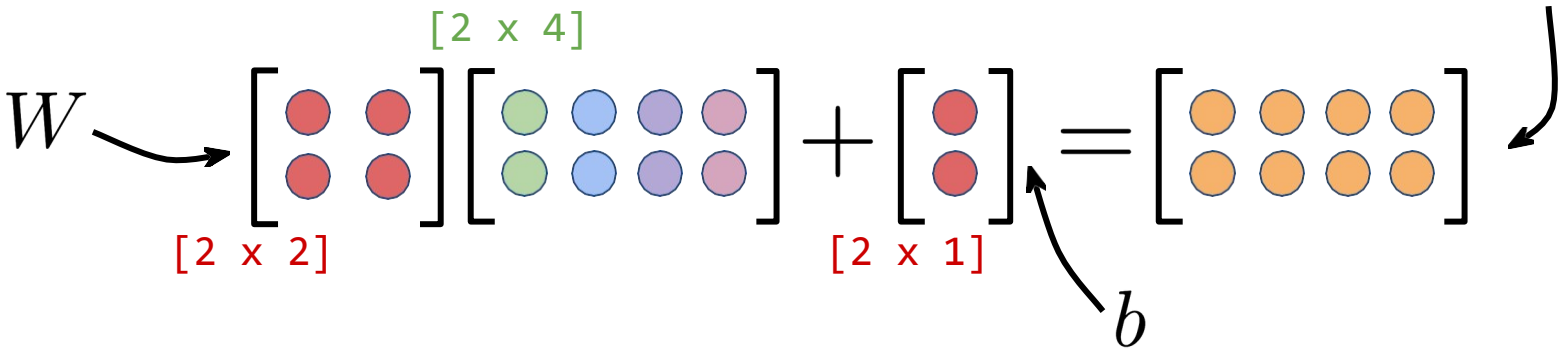
Processing several instances makes the overall processing much faster!

Batch Processing



vs

scores for all examples
per column



Batch Processing

Batch gradient descent leads to more “stable” updates as well - the direction towards the optimal parameters is computed after looking at all examples, instead of just one!

Batch Processing

What if you had a few outliers (bad examples)

- SGD will cause the parameters to drift farther from their optimal values when the update loop goes over these outliers
- Batch GD will drown out the effect of the outliers since there are many more good examples

Batch Processing

But

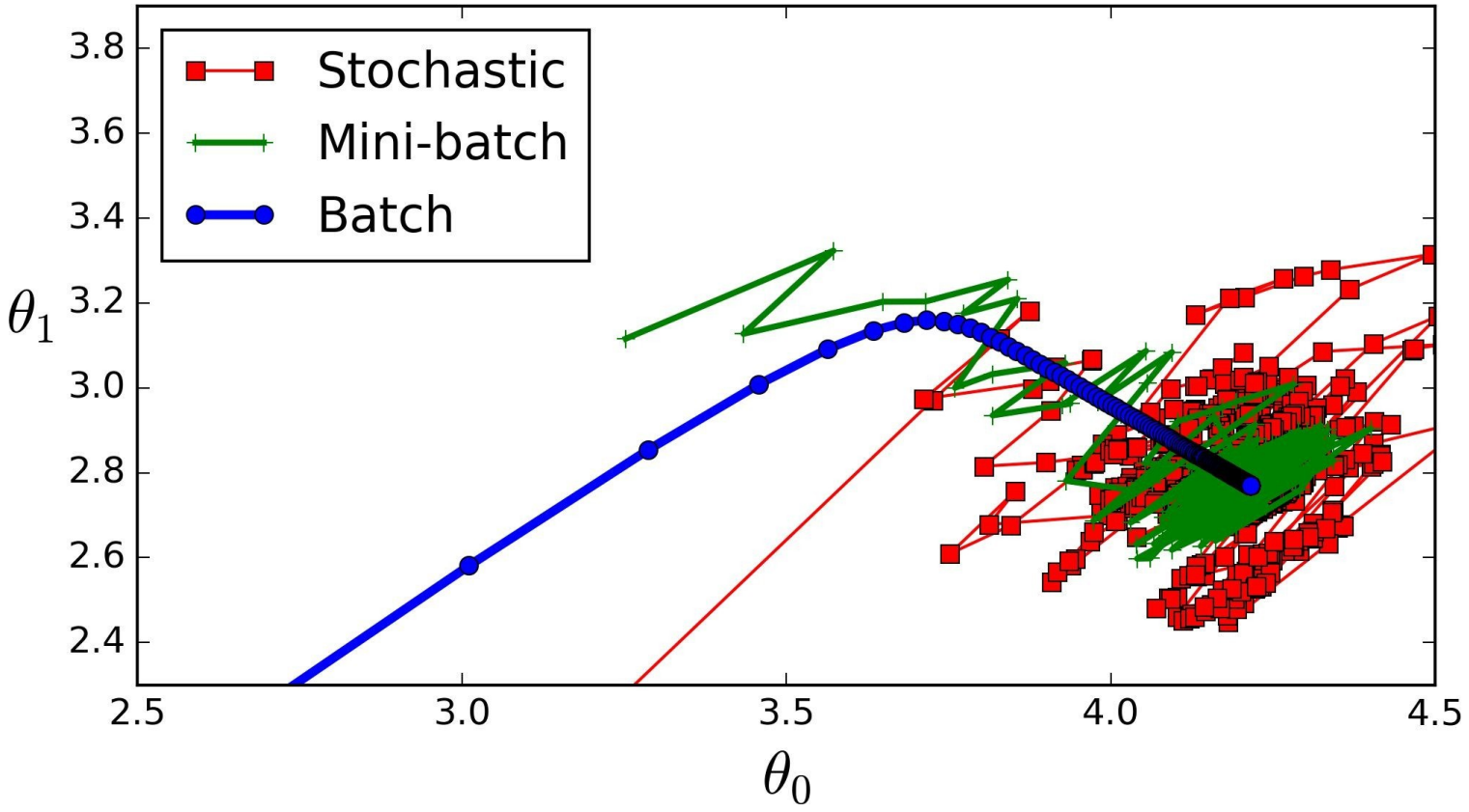
- Batch GD requires us to look over the entire dataset before making any progress - so it's much slower
- The entire dataset may not even fit in memory, so making the overall code efficient would be more difficult

Batch Processing

Solution

- **Minibatch SGD**: Perform updates after looking at a “minibatch” (e.g. 32 data points)
- Much faster than Batch GD, but largely avoids the issues with SGD

Batch Processing



https://stats.stackexchange.com/a/153535

Padding

Practical Considerations

Padding

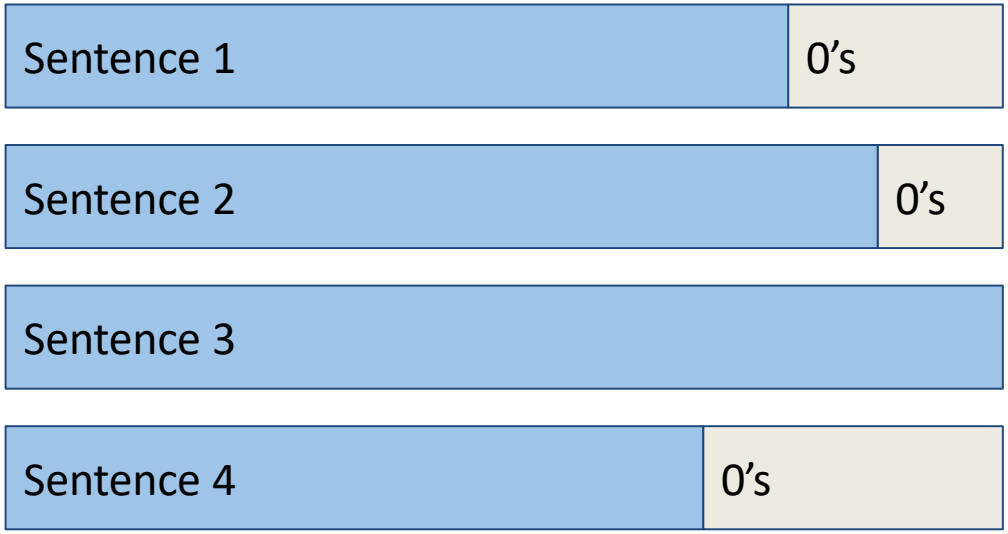
- Input sentences are of varied length
- Need a fixed length to define a fixed size of weight matrices

Padding

Solution: Pad smaller sentences with 0's

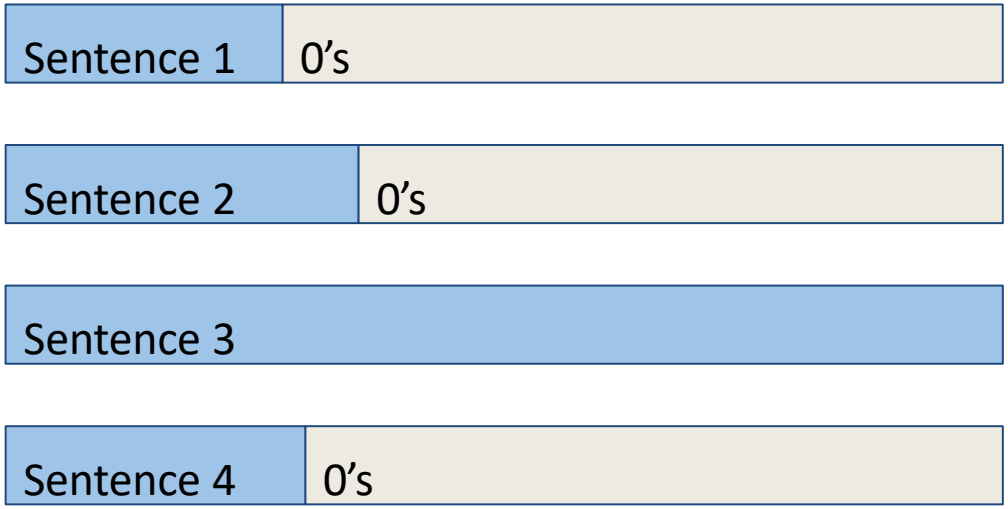
Padding

Solution: Pad smaller sentences with 0's



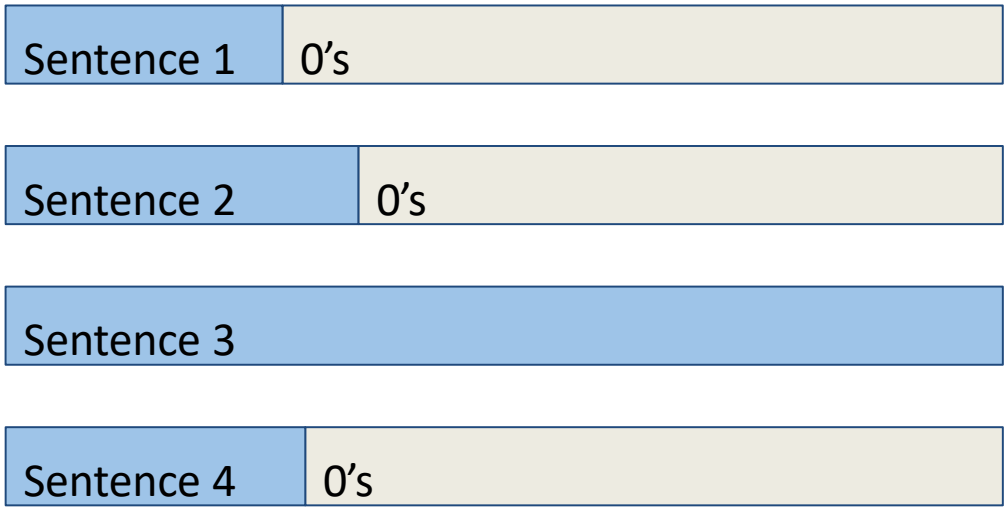
Padding

Problem: What if one sentence is very long in a batch?



Padding

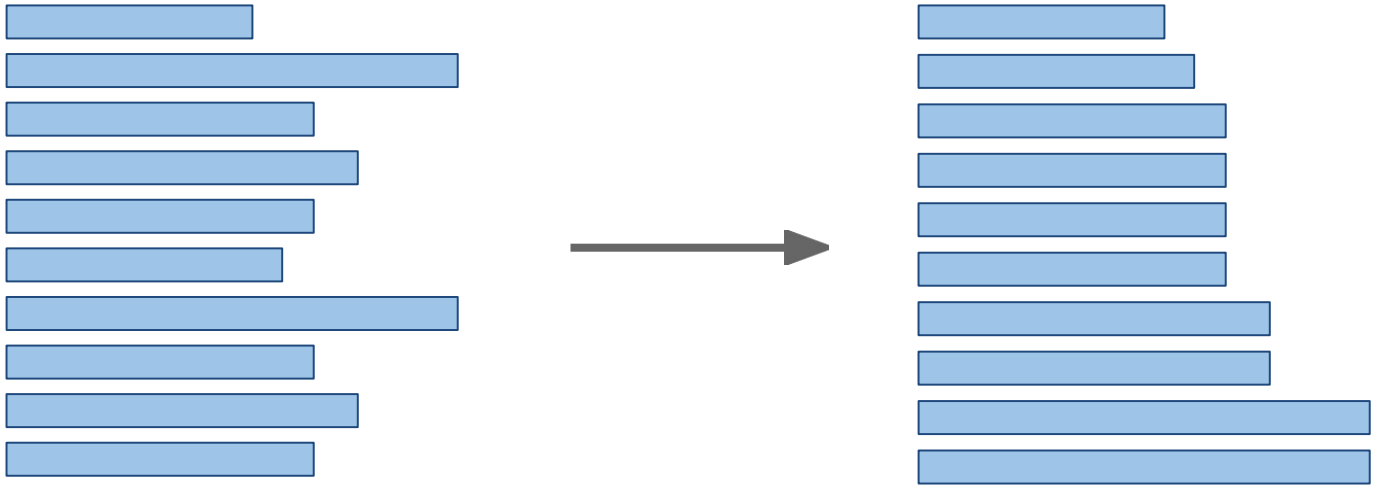
Problem: What if one sentence is very long in a batch?



A lot of wasted computation!

Padding

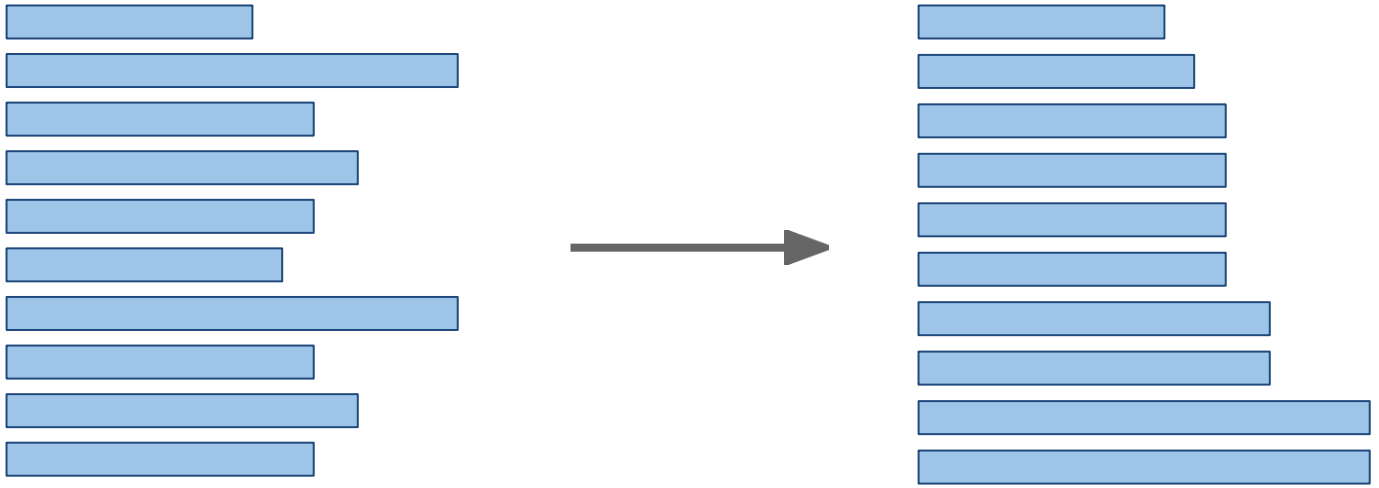
- Alternatively, sort all sentences by length
- Create minibatches by putting sentences of similar length together



- Limit wasted computation in every minibatch

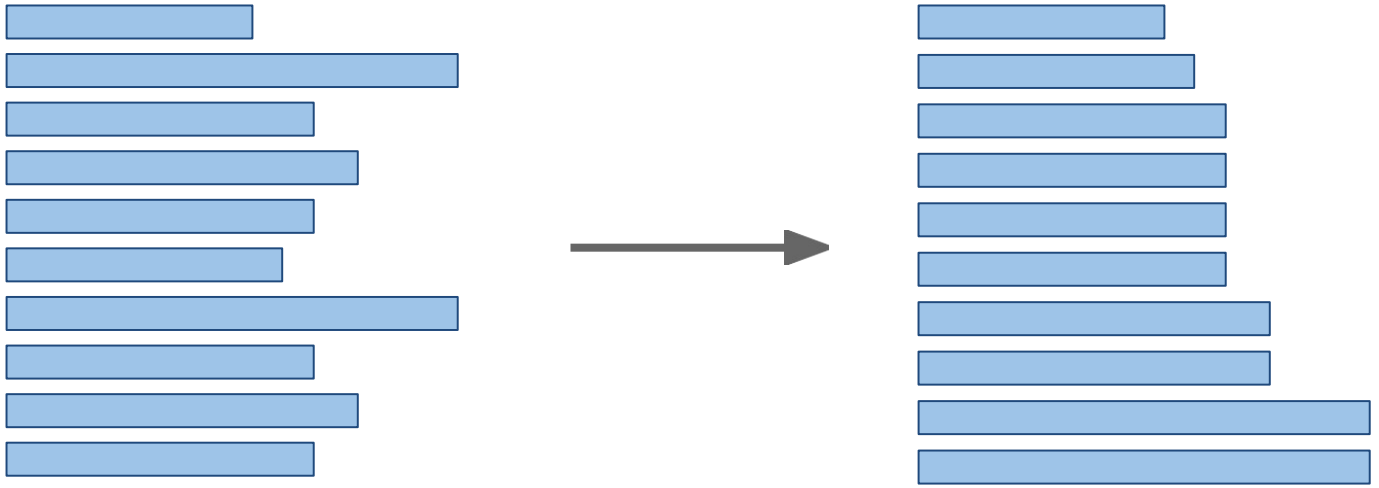
Padding

Q: Any problems with this solution?



Padding

Q: Any problems with this solution?



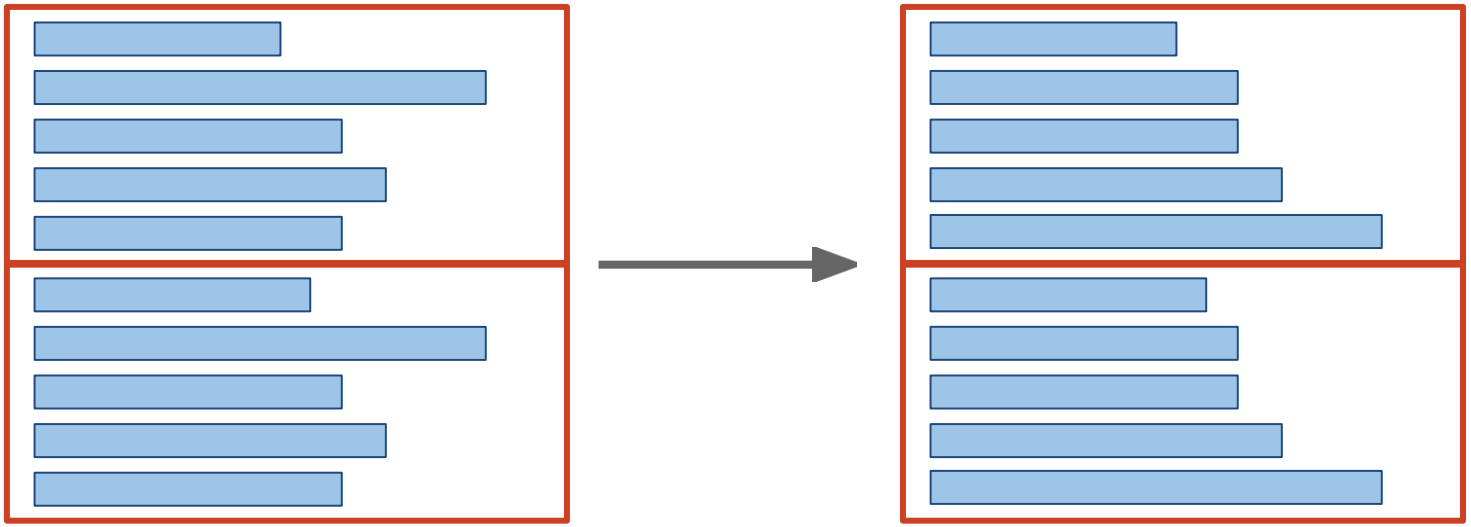
A: Yes! We are inducing a bias so that the model sees all short sentences early and all long sentences later

Padding

Solution: Sort sentences in a **maxi-batch**

Padding

Solution: Sort sentences in a **maxi-batch**



Now choose minibatches from each maxibatch

Class Imbalance

Practical Considerations

Class Imbalance

Often times real world data don't have an equal representation of all the classes, e.g. 80% positive data and 20% negative data

Class Imbalance

Often times real world data don't have an equal representation of all the classes, e.g. 80% positive data and 20% negative data

While training such data, you will reach 80% accuracy very quickly!

Class Imbalance

Often times real world data don't have an equal representation of all the classes, e.g. 80% positive data and 20% negative data

While training such data, you will reach 80% accuracy very quickly!

But this is not a good thing - the reason you reach this high of an accuracy so quickly is that the model learns to predict "positive" for all input instances

Class Imbalance

Solution: Many possible solutions - one of the easiest thing to try is to weight the losses differently for each class

Class Imbalance

Solution: Many possible solutions - one of the easiest thing to try is to weight the losses differently for each class

Updates from the positive instances should be small - and updates from the negative instances should be large (preferably proportional to the imbalance)

Class Imbalance

Solution: Another solution is the *undersample* the majority class; Reduce the number of examples in the *positive* class to match the number is the *negative* class.

Class Imbalance

Solution: Another solution is to *undersample* the majority class; Reduce the number of examples in the *positive* class to match the number is the *negative* class.

A disadvantage here is that we are throwing away a lot of data - and data is precious!

Class Imbalance

Solution: A third solution is to *oversample* the minority class; Increase the number of examples in the *negative* class to match the number is the *positive* class.

Doesn't always work super well, since there is a lot of redundancy in the minority class

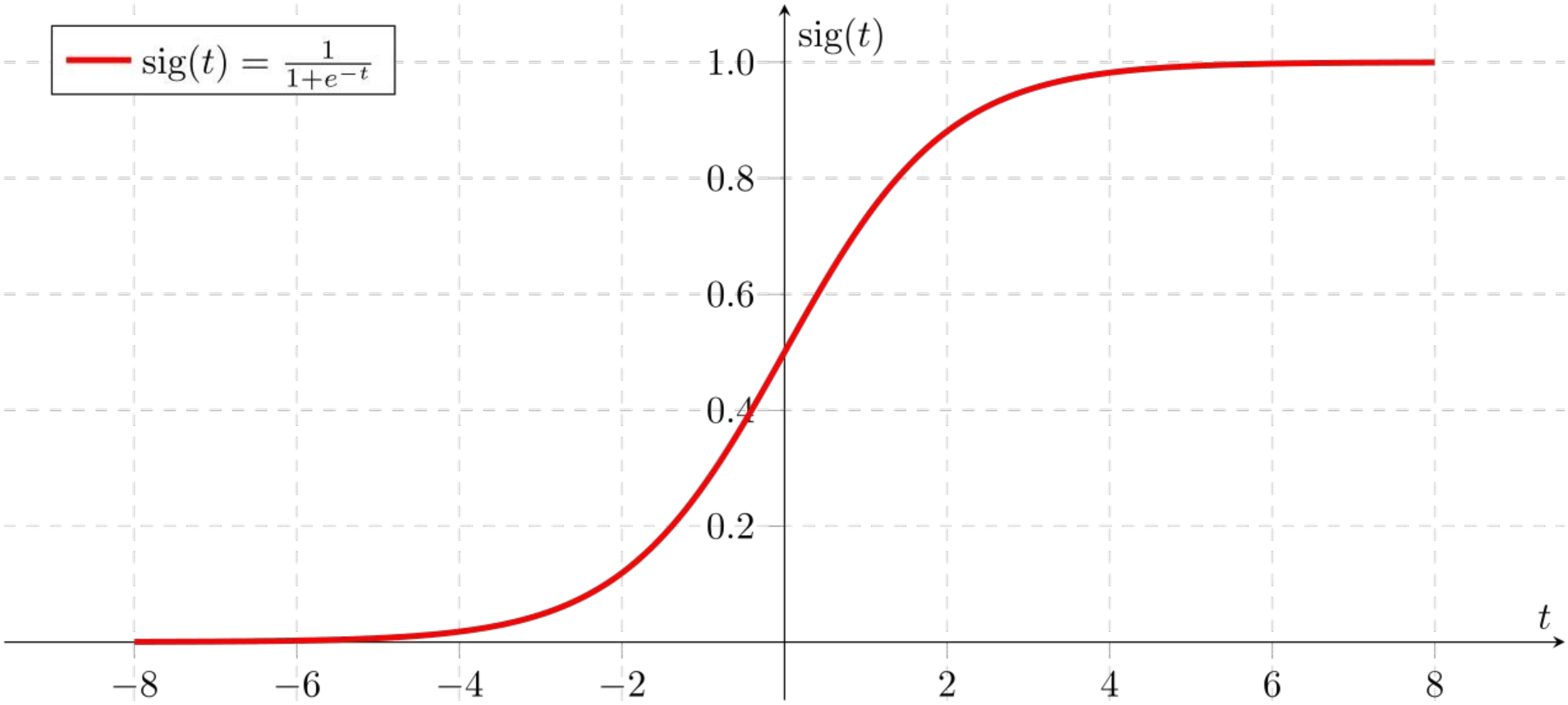
Data Normalization

Practical Considerations

Data Normalization

- Features in the data come at different scales
- Large feature values when multiplied with random weights result in larger values
- Activation functions like sigmoid result in the saturation of these neurons
- The optimization stops working for those parameters since the gradient becomes very small or very close to zero

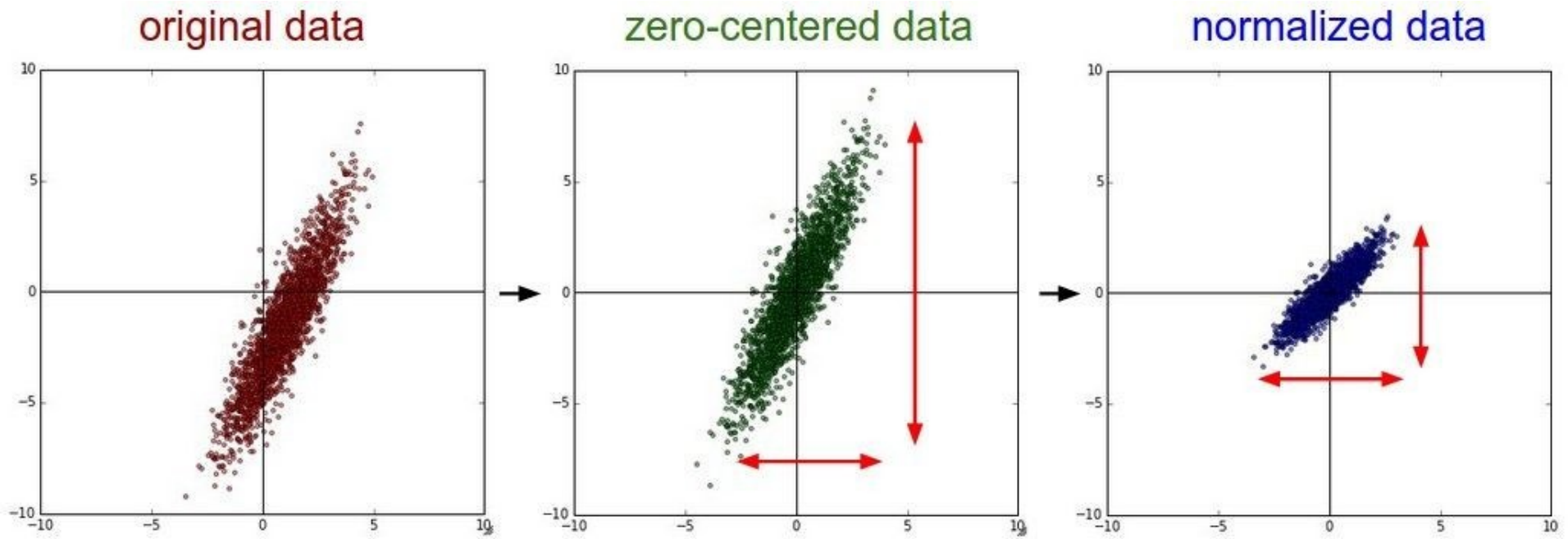
Data Normalization



Data Normalization

- Normalizing the data before training the model is one way to solve this problem
- A common method of data normalization is to first **zero center** the data by subtracting the **mean** across every *feature* in the data, and then **divide** the resulting value with **standard deviation**
- This results in the data having *unit variance*

Data Normalization



https://cs231n.github.io/neural-networks-2/#datapre

Parameter Initialization

Practical Considerations

Parameter Initialization

- We've learned that we need to move in the direction of the gradient to reach the minimum value for a given loss function
- But where do we start?

Parameter Initialization

- Initial values of `W` and `b` dictate where in the terrain we begin
- If we start near a minima, we can optimize very quickly - If we start too far, it may take a long time to find a good model
- We may even start near a local minimum and never find the global minimum for a given function

Parameter Initialization

- Zero initialization?
- Random initialization?
- Something more complicated?

Parameter Initialization

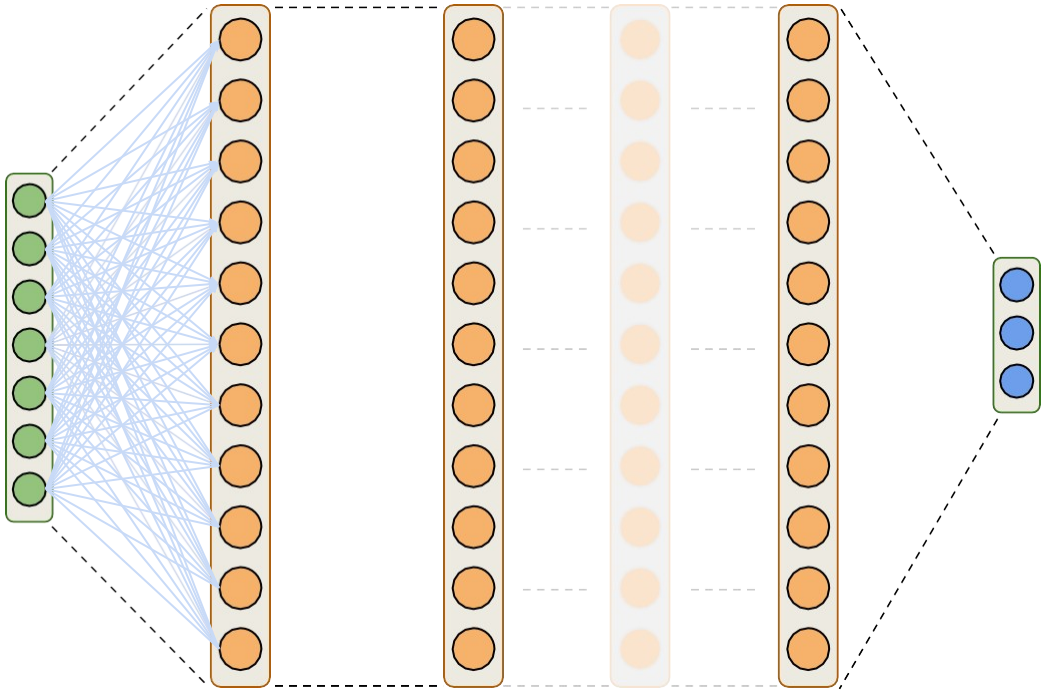
- Zero initialization
- Random initialization
- Something more complicated:
  - Gaussian distributed
  - Xavier Initialization

Zero Initialization

Q: Why should we never use zero initialization with neural networks?

Zero Initialization

Q: Why should we never use zero initialization with neural networks?



Zero Initialization

Q: Why should we never use zero initialization with neural networks?

A: All the neurons see the same input - and with the same weight matrices, they will make the exact same decisions!

Random Initialization

- Randomly initialize weights close to zero
- Avoids symmetry in the network by having different random weights

Xavier Initialization

There is a better way to initialize weight matrices other than random numbers: **Xavier initialization**

Xavier Initialization

There is a better way to initialize weight matrices other than random numbers: **Xavier initialization**

Different for each layer - depends on the number of connections coming in and going out!

Xavier Initialization

Usually when we pick random numbers, we pick them from a uniform distribution

Xavier Initialization

Usually when we pick random numbers, we pick them from a uniform distribution

Xavier initialization says we should pick the random numbers from a distribution with zero mean and the following variance:

$$Var(W) = \frac{2}{n_{in} + n_{out}}$$

Xavier Initialization

Works very well in practice - was actually an enabler in training deeper networks at some point in time!

Dropout

Practical Considerations

Overfitting

- Humans tend to believe what comes frequently in their lives or daily routine
  - a recurrent advertisement make us believe it
  - in return, we are less open to any new information
- Similarly, machines tend to overfit what they have seen during training
  - result in less robust model to any unseen scenario

Dropout

- Neural network models tend to overfit the training data
  - performs poorly on unseen data

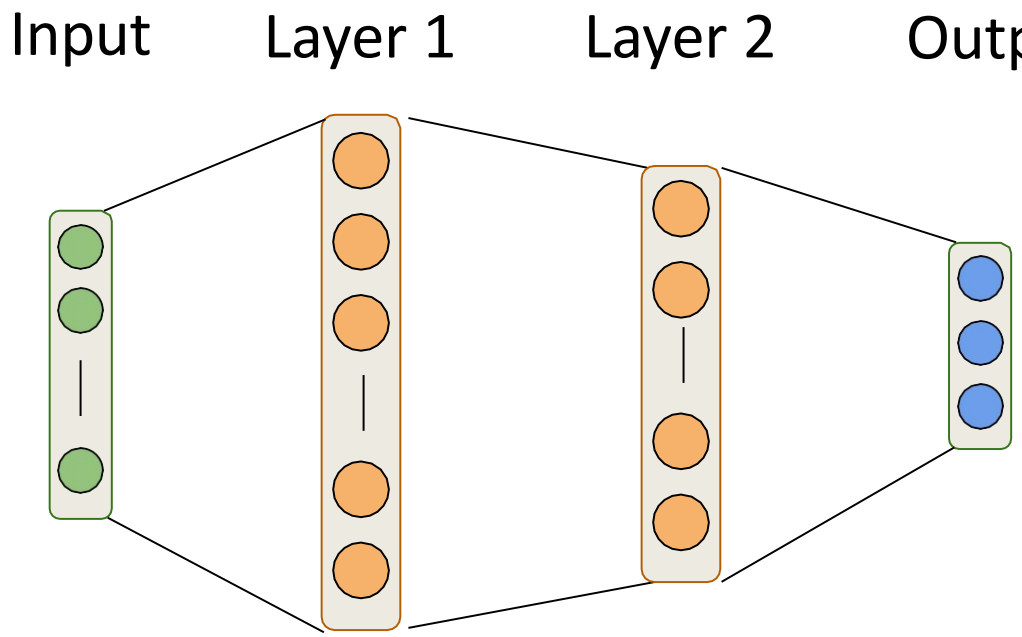
- How can we expose our network to diverse scenarios?
- Can we make the training of the model more robust and improve generalization?

Dropout

"Randomly drop neurons from the network during training"

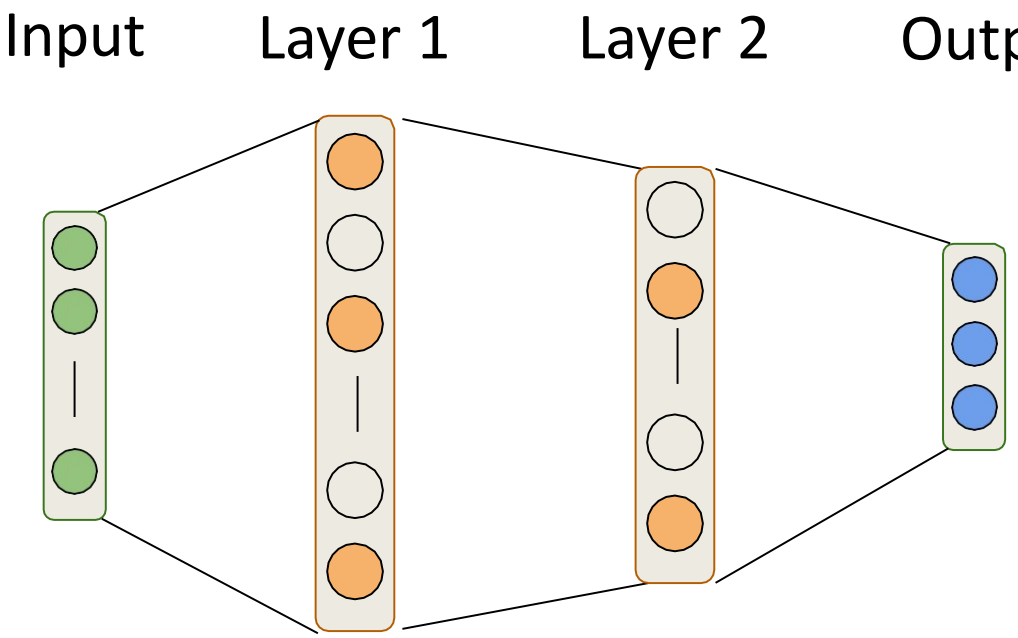
- **Intuition:** give network a wider class of scenarios to tackle
- By dropping a few neurons, we are essentially forcing the model to learn in scenarios when some information is missing
- Some other neuron(s) have to step up to handle this situation

Dropout



- Let’s say we want to randomly drop a few neurons from every layer

Dropout



- In other words, some information in the network is missing
- Now model has to learn to reduce loss with fewer neurons
- This improves generalization of the model

Dropout

- Algorithmically, say we want to apply dropout of p=0.5 on the complete network

- At train time, for every layer in every iteration:
  - For every neuron
    - predict a random number between 0 and 1
    - if number is less than 0.5, drop that neuron and its connections

- At test time, **always use the complete network**
  - compensate for missing activations during training by reducing all activations by the factor p

Dropout

- Frequently used while training models
  - specially when training data is small
  - in machine translation, it gives a BLEU gain of up to 3 points on small data

Dropout

Implementing in Keras

```
model = Sequential()
model.add(Dense(100, input_shape=(len(x_train[0]),), activation='sigmoid'))
model.add(Dropout(0.2))
model.add(Dense(50, activation='sigmoid'))
model.add(Dropout(0.3))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['acc'])

model.fit(x_train, y_train, verbose=True, epochs=5, validation_split=0.05)
```

Dropout

Implementing in Keras

```
model = Sequential()
model.add(Dense(100, input_shape=(len(x_train[0]),), activation='sigmoid'))
model.add(Dropout(0.2))
model.add(Dense(50, activation='sigmoid'))
model.add(Dropout(0.3))
model.add(Dense(num_classes, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['acc'])

model.fit(x_train, y_train, verbose=True, epochs=5, validation_split=0.05)
```

Ensembles

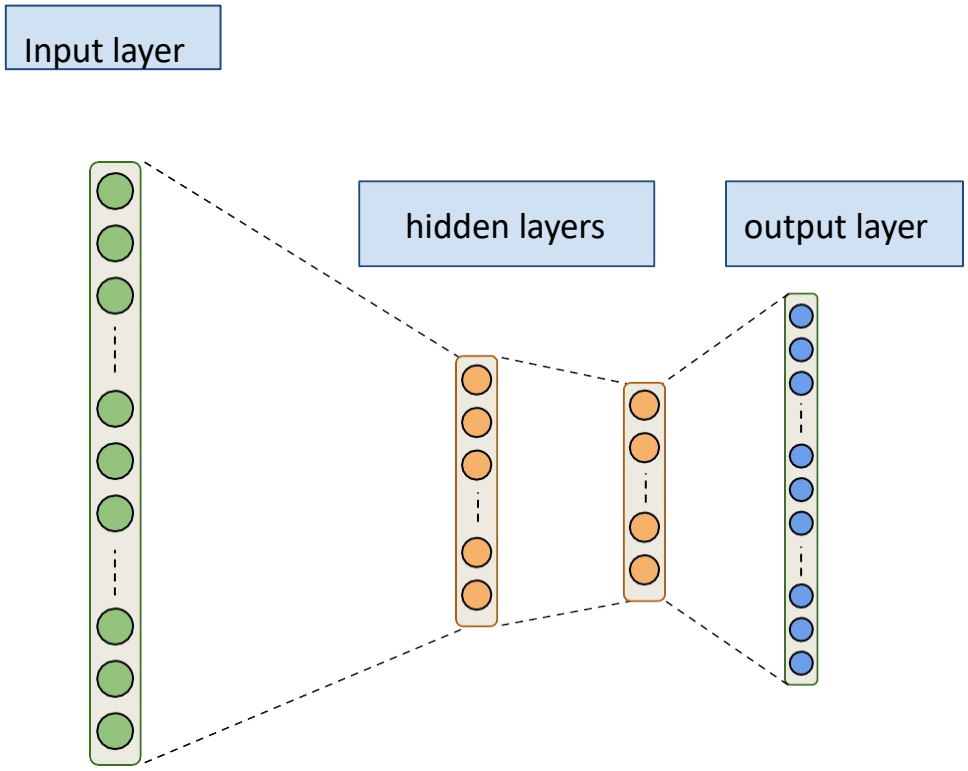
Practical Considerations

Ensemble

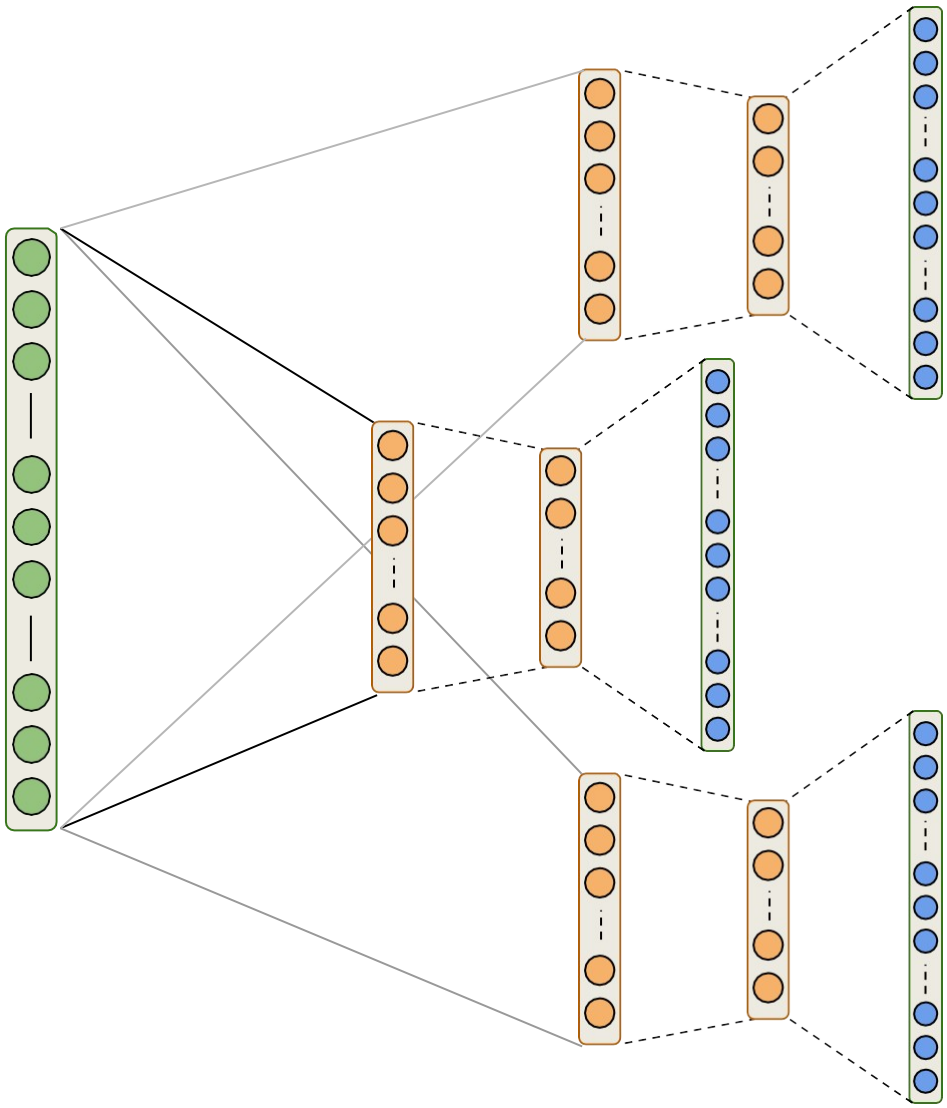
- Neural network models are trained on random weights, thus result in different models
- Every model may have specialized in slightly different aspects of the data
- In ensemble, we combine several trained models at test time

Ensemble

Single model



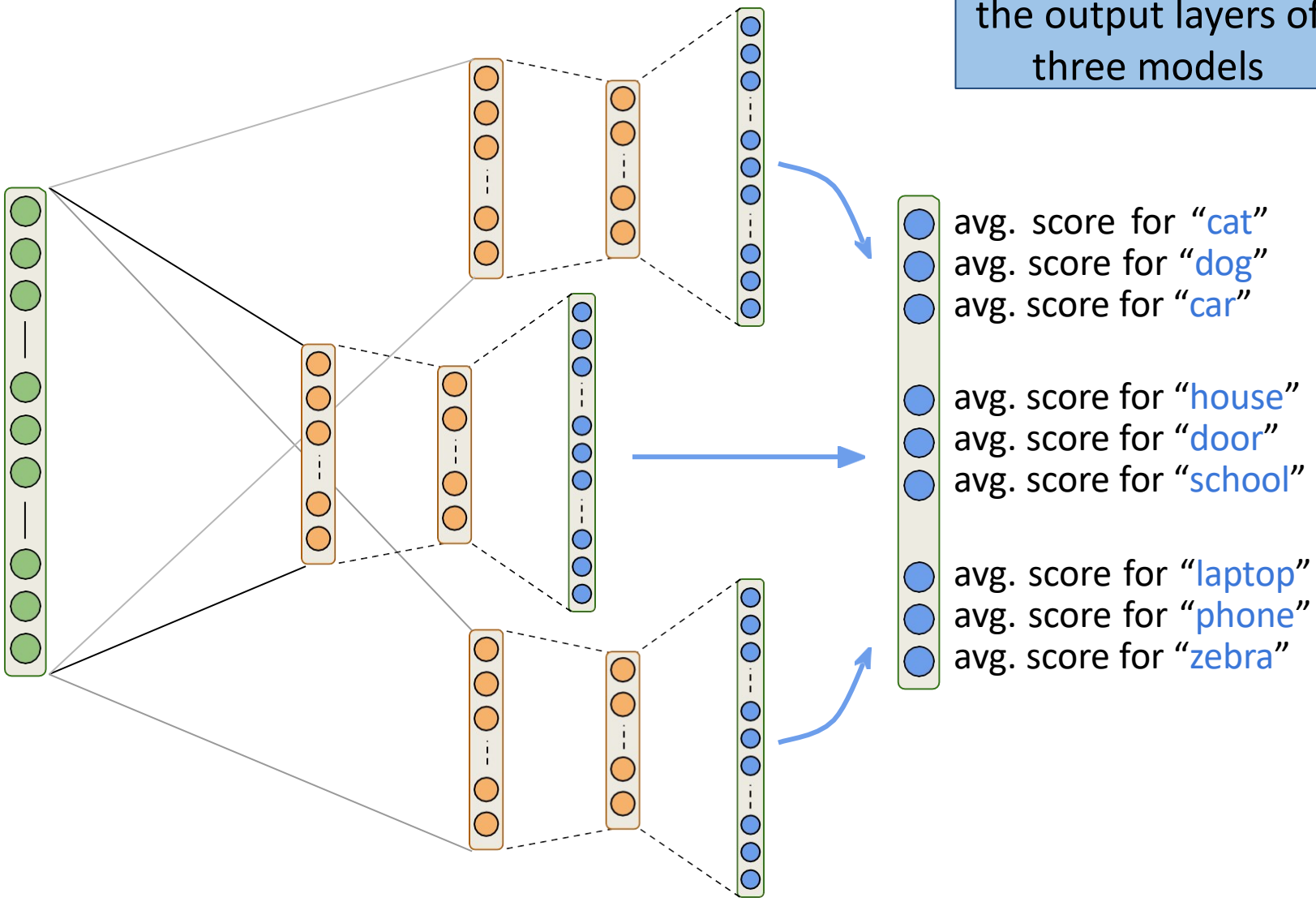
Ensemble



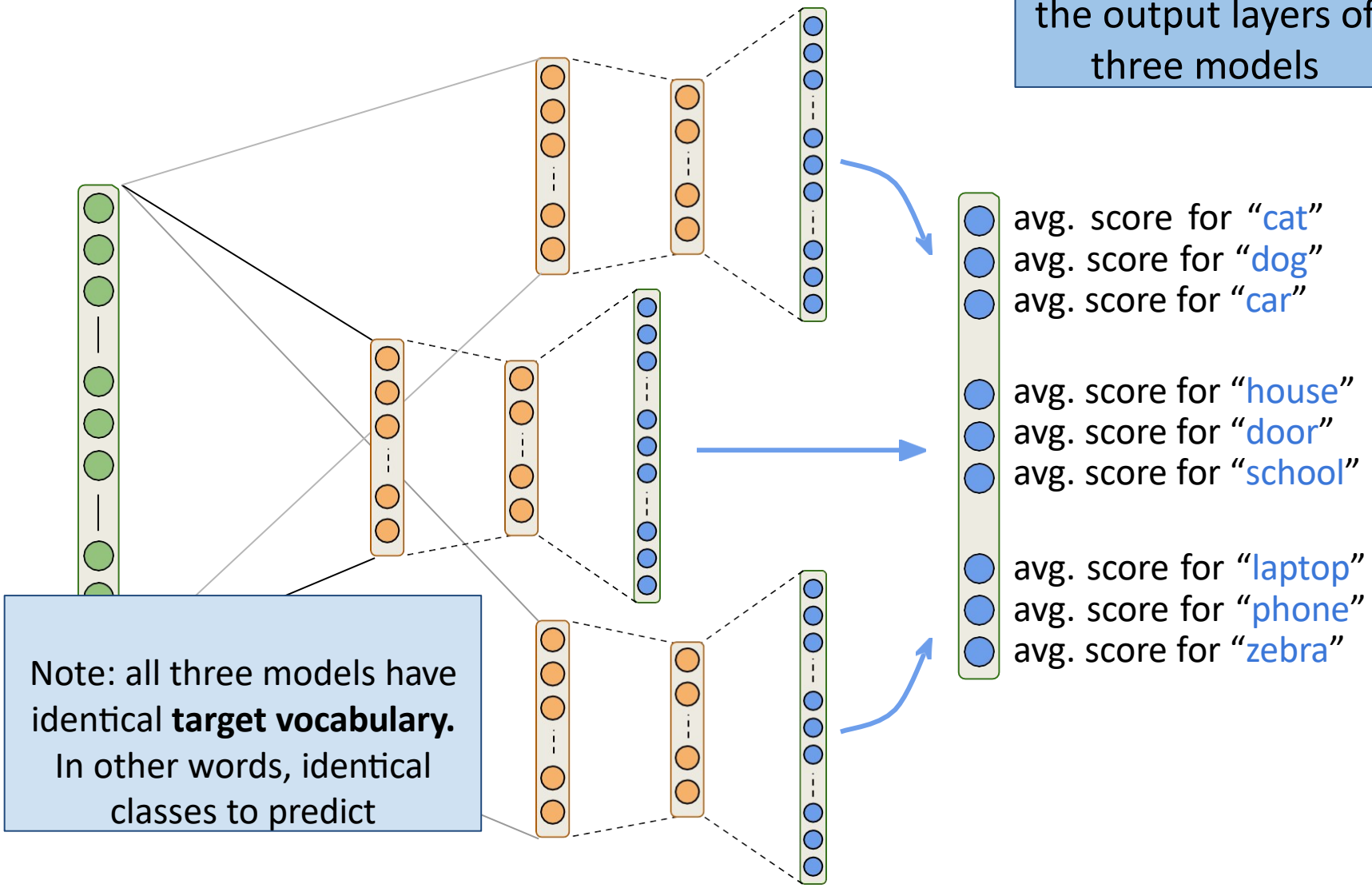
Three models

Ensemble

Average scores of the output layers of three models



Ensemble


Average scores of the output layers of three models
avg. score for "cat"
avg. score for "dog"
avg. score for "car"
avg. score for "house"
avg. score for "door"
avg. score for "school"
avg. score for "laptop"
avg. score for "phone"
avg. score for "zebra"
Note: all three models have identical **target vocabulary.** In other words, identical classes to predict

Ensemble

- Ensemble can also be done on:
  - **models trained on different datasets** but share identical target vocabulary (labels/classes)
  - different checkpoints of the same model
    - e.g. models after epoch = 5, 10, 15
  - beneficial to combine models trained on different data
- In machine translation, ensemble gives a performance improvement of 1-2 BLEU points
- Slow for real time processing

Residual Connections

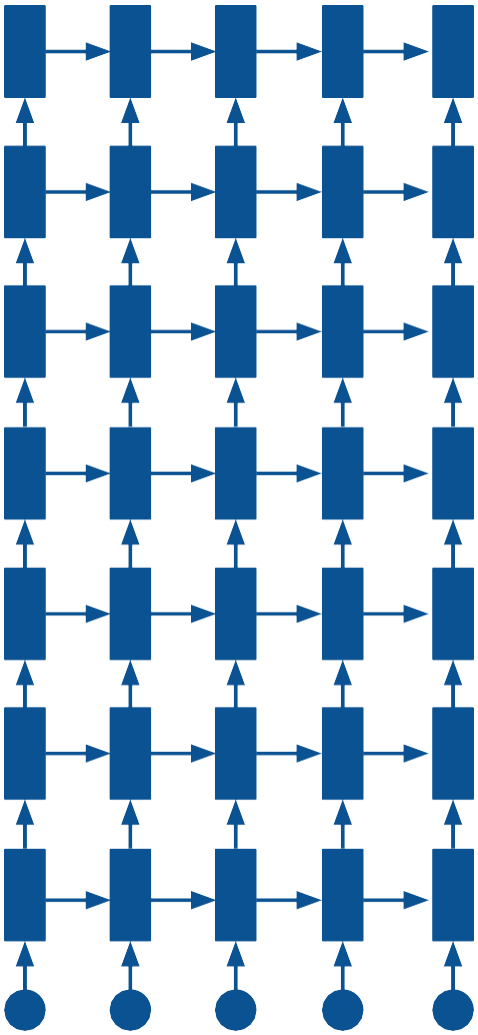
Practical Considerations

Residual Connections

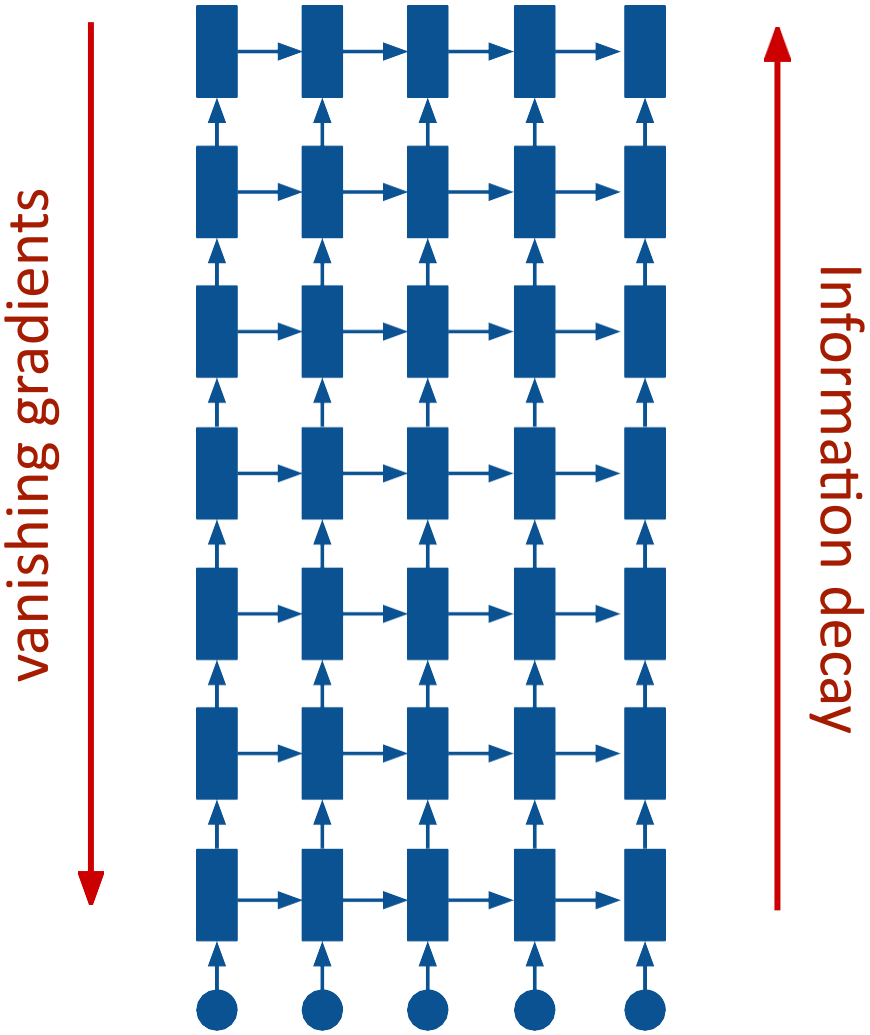
Deeper the better

- As we go deeper in the network:
  - model starts forgetting the input to the network
  - gradient from higher layers to initial layers starts diminishing (vanishing gradient)
- Residual networks keep shortcut connections between different layers
  - in practice, several combinations of connections are used between layers

Residual Connections

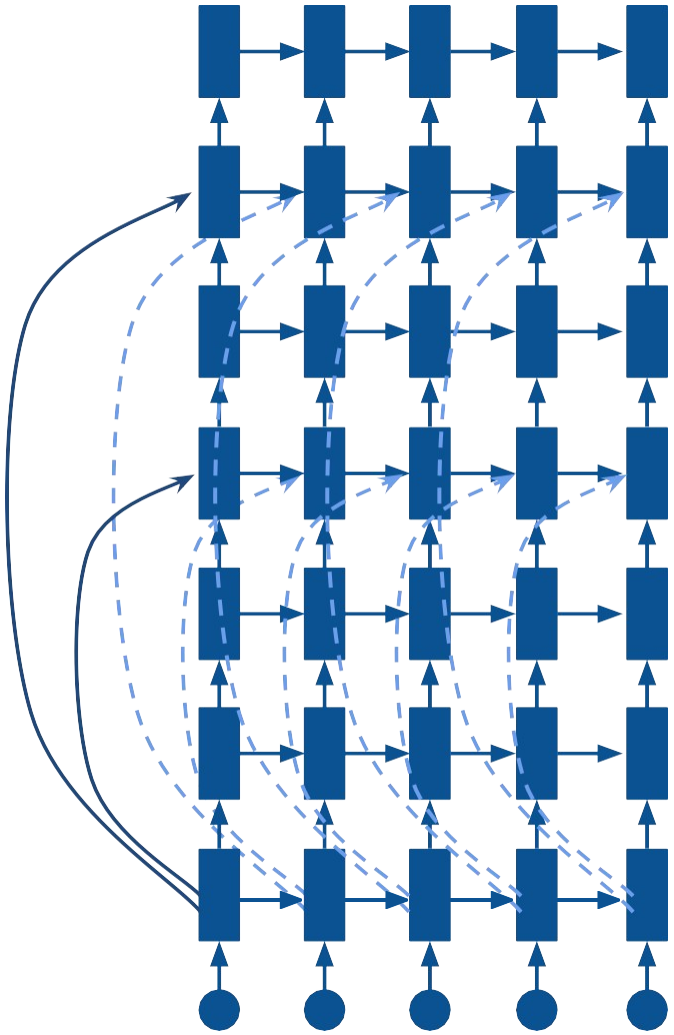


Residual Connections



Residual Connections

h=*g*(Wx + b) + **x**



Shortcut connections from the embedding layer to higher layers is one common implementation

Note that we are increasing the overall number of parameters in the network!

Batch Normalization

Practical Considerations

Batch Normalization

- Normalization of input layer improves the training of the model
- We can apply a similar normalization to all the

  layers of the model
- This speeds up the training and avoids the issues

  caused by bad initialization of the parameters

Batch Normalization

- For every layer, batch normalization ensures that the values of the weights remain under some threshold, thus avoiding the issue of neuron saturation
- It also provides a safeguard against overfitting - since it forces the activations to follow a *gaussian distribution*

Gradient Accumulation

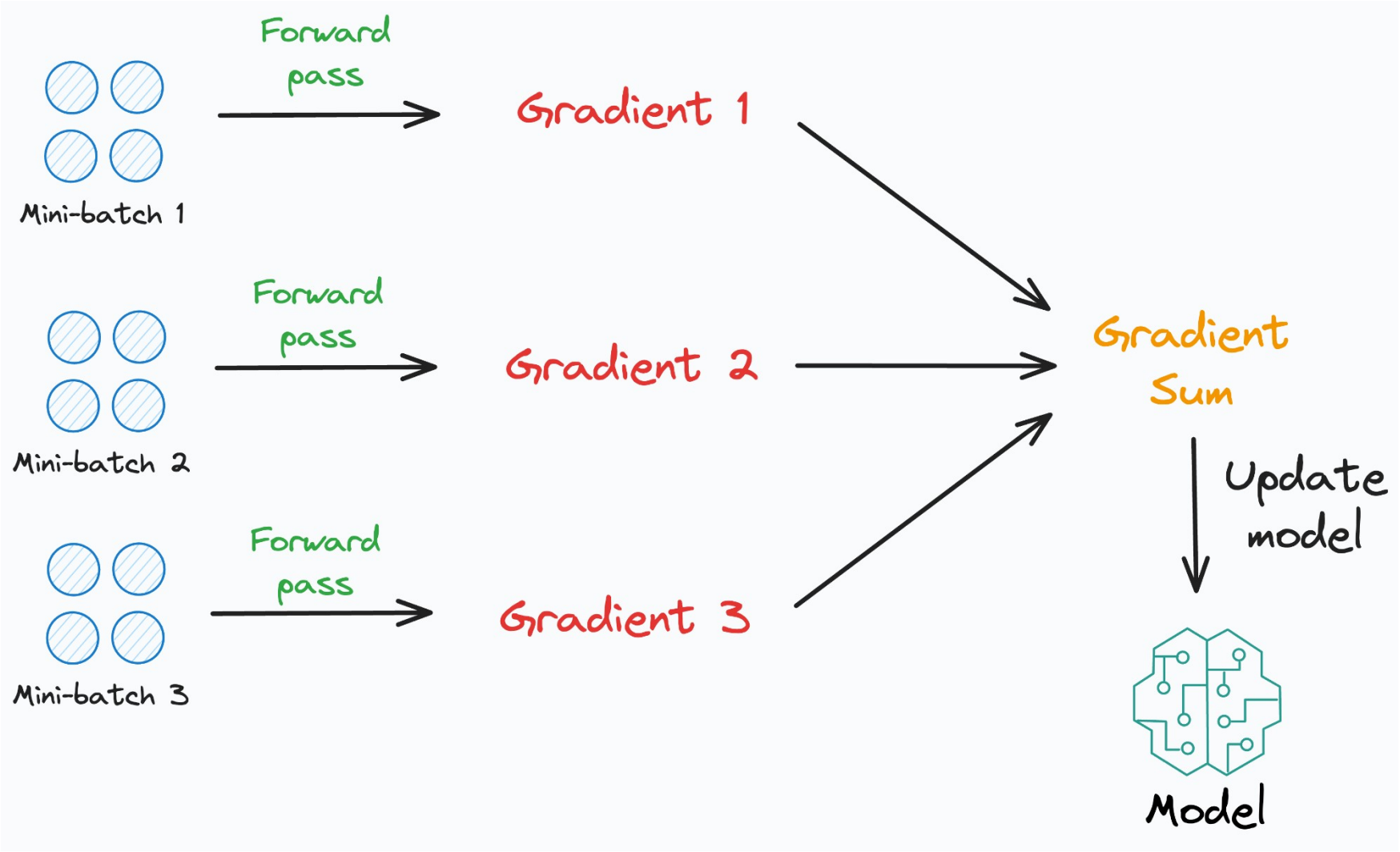
Let's say we want to train an 1B parameter model and loading the model is possible but we run out of memory during training. How can we solve this?

What if we have low resources available and cannot fit large batch sizes into memory?

Gradient Accumulation

- Gradient accumulation allows us to effectively simulate a larger batch size without actually processing a large batch at once.
- It works by accumulating gradients over multiple mini-batches before updating the model parameters

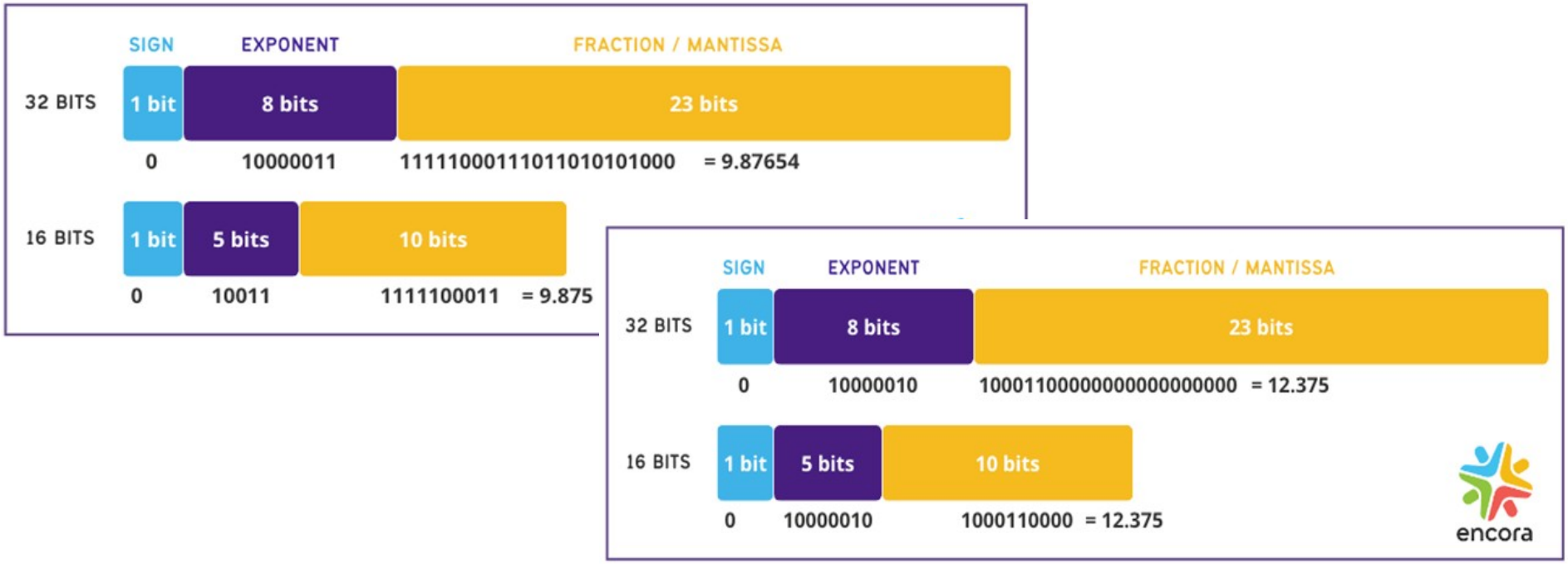
Gradient Accumulation



Gradient Accumulation: Trade-offs

- can increase the training time due to the need to process multiple mini-batches.
- Larger batch sizes can lead to faster convergence but might require more memory and computational power.
- Smaller batch sizes can be more stable and provide better generalization but might take longer to train.

Quantization

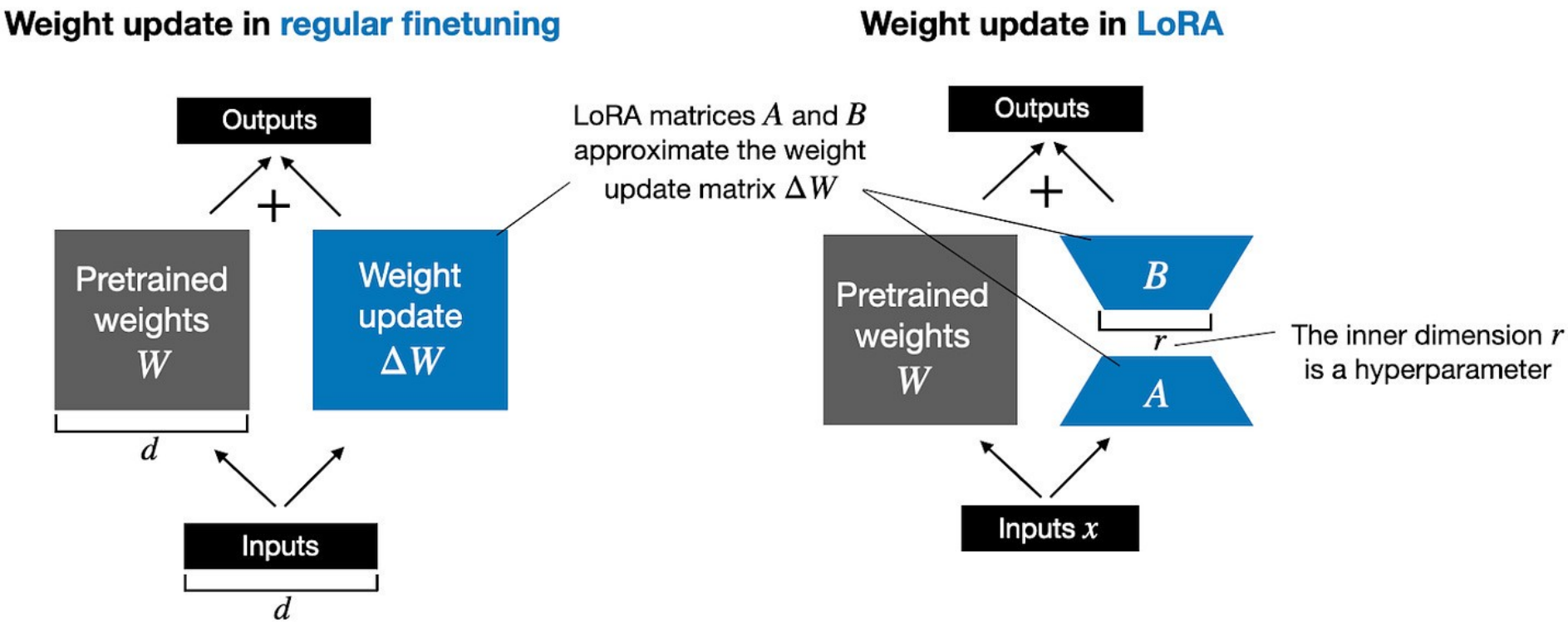


- reducing the precision of model weights and activations, typically from 32-bit floating point to 8-bit integers.
- may **lose** some important details, which can reduce the model's accuracy.
- When can we use quantization?

Quantization

- reducing the precision of model weights and activations, typically from 32-bit floating point to 8-bit integers.
- may **lose** some important details, which can reduce the model's accuracy.
- When can we use quantization?
- During inference, to optimize memory and work with bigger models

LoRa: **Low-Rank Adaptation**

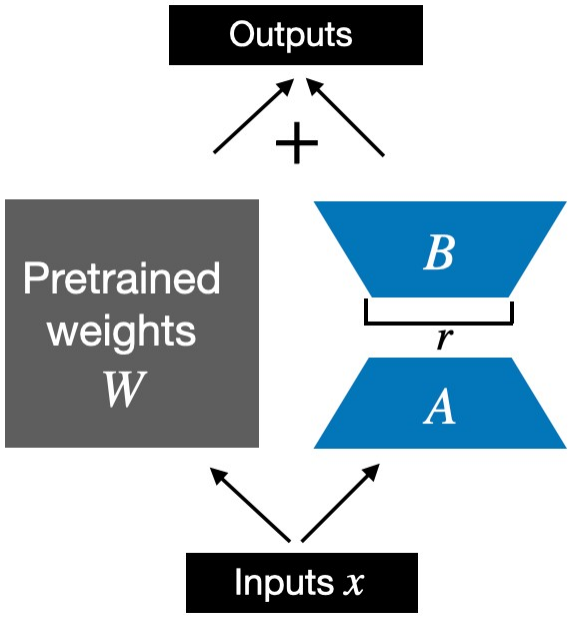


$$W_{updated} = W + \Delta W$$

$$W_{updated} = W + A.B$$

https://magazine.sebastianraschka.com/p/lora-and-dora-from-scratch
https://www.twinko.ai/app/llm-gpu-memory-calculator

How does LoRA save GPU memory?

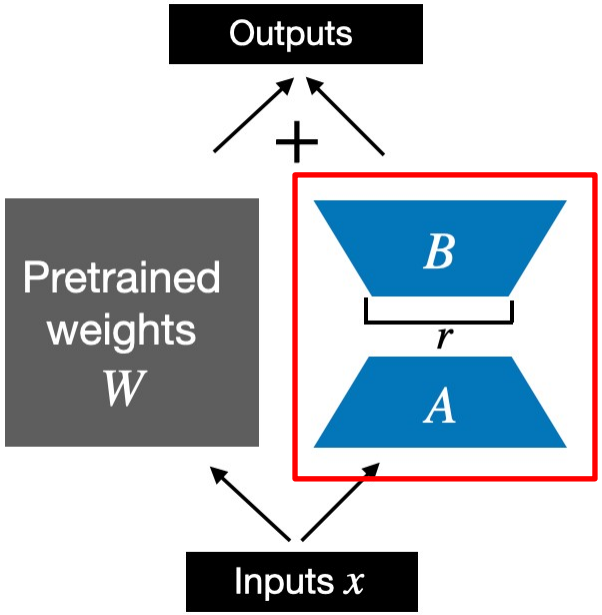


Let ***W*** be a 1,000×1,000 matrix. Then the weight update matrix ***ΔW*** in regular finetuning is a 1,000×1,000 matrix as well.
In this case, ***ΔW*** has 1,000,000 parameters.

If we consider a **LoRA** rank of 2, then ***A***=1000×2 matrix, and ***B***=2×1000 matrix, we have 2×2×1,000 = 4,000 parameters that we need to update when using LoRA.

https://magazine.sebastianraschka.com/p/lora-and-dora-from-scratch
https://www.twinko.ai/app/llm-gpu-memory-calculator

How does LoRA save GPU memory?



x.(W+A.B) = x.W + x.A.B

We can keep the LoRA weight matrices separate - we don't have to modify the weights of the pretrained model at all, as we can apply the LoRA matrices on the fly.

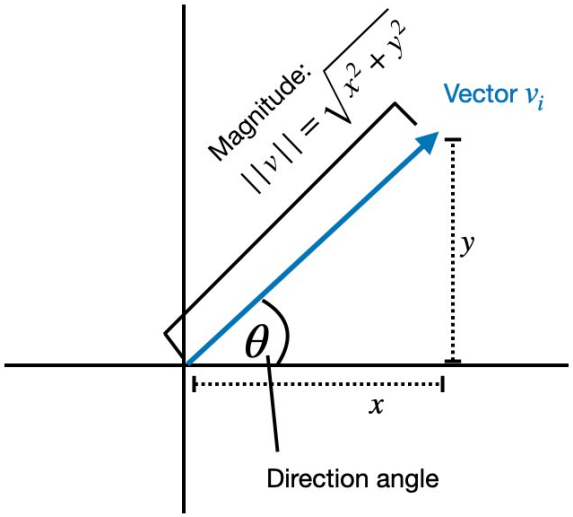
Why is this important?

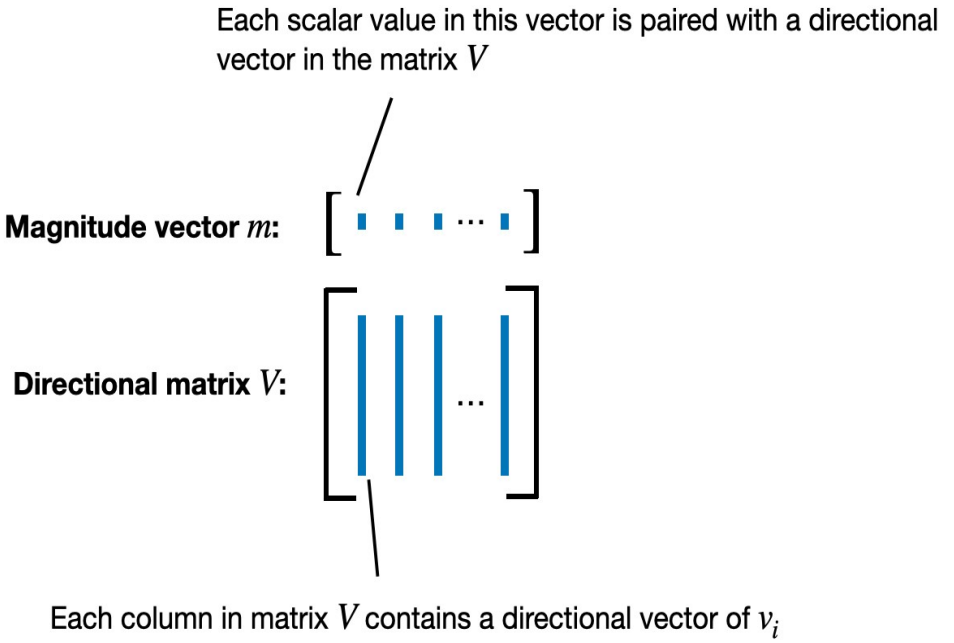
If you are considering hosting a model for multiple customers. Instead of having to save the large updated models for each customer. *You only have to save a small set of LoRA weights alongside the original pretrained model.*

https://magazine.sebastianraschka.com/p/lora-and-dora-from-scratch
https://www.twinko.ai/app/llm-gpu-memory-calculator

DoRA -Weight-Decomposed Low-Rank Adaptation

Decompose a pretrained weight matrix into a **magnitude vector (m)** and a **directional matrix (V)**

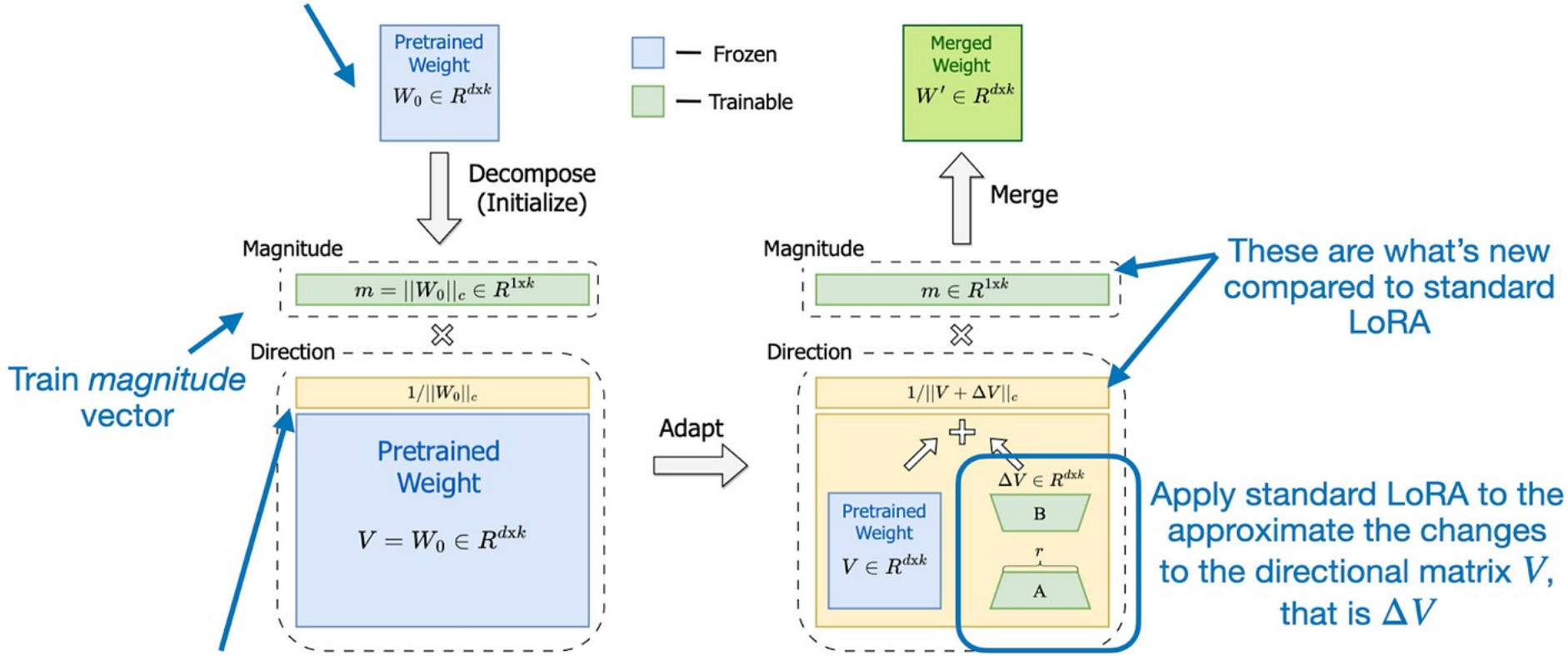




https://magazine.sebastianraschka.com/p/lora-and-dora-from-scratch

https://www.twinko.ai/app/llm-gpu-memory-calculator

At each step you take, there are **two critical decisions: the direction in which you walk**, and **how far you go** in that direction. Researchers discovered that the way these decisions, direction and magnitude, are taken in LoRA differs from how they're taken during Full Fine-Tuning and tried to bridge that gap.



where the decomposition is

$$W_0 = m\frac{V}{\|V\|_c} = \|W_0\|_c\frac{W_0}{\|W_0\|_c}$$

hence the $1/\|W_0\|_c$