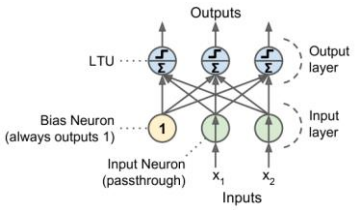
Module 6 Deep feedforward networks

- Simple artificial neural network:
  - One or more binary inputs and one binary output
  - Active its output when more than a certain number of its inputs are active
- Linear threshold unit (LTU)
  - Inputs are numbers (not binary)
  - Each input connection is associated with a weight
  - Computes a weighted sum of its inputs and applies a step function to that sum
- Perceptron
  - It is a single layer of LTUs
  - The input neurons output whatever input they are fed
  - A bias neuron, which just outputs 1 all the time
  - 

- Training of perceptron
  - Inspired by Hebb's rule: when a biological neuron often triggers another neuron, the connection between tow neurons grows stronger
  - Feed one training instance x to each neuron j at a time and make its prediction $\hat{y}$, and then update the connection weights
  - $\hat{y_j} = \sigma\left(w_j^T x + b\right)$
  - $j(w_j) = crossentropy(y_j, \hat{y_j})$
  - $w_{i,j}^{next} = w_{i,j} - \eta \frac{\partial J(w_j)}{w_i}$
- Multi-layer perceptron (MLP)
  - The weakness of perceptron: incapable of solving some trivial problems, e.g., XOR classification. If we minimize $J(w) = \frac{1}{4}\left(\hat{y}(x) - y(x)\right)^2$, we obtain $w_1 = 0, w_2 = 0, b = \frac{1}{2}$. The model outputs 0.5 everywhere
  - It can be solved by stacking multiple perceptrons
  - The resulting network is called a multi-layer perceptron (MLP) or deep feedforward neural network
- Feedforward Neural Network Architecture
  - It is composed of
    - One input layer
    - One or more hidden layers
    - One final output layer
  - Every layer except the output layer includes a bias neuron and is fully connected to the next layer
  - The length of the chain gives the depth of the model

- Feedforward neural network - Cost function
  - Cross-entropy (minimize the negative log-likelihood)
  - $cost(y, \hat{y}) = -\sum_j y_j \, log(\hat{y}_j)$
- Gradient-based learning
  - The most significant difference between the linear models and feedforward neural network is that the non-linearity of a neural network causes its cost functions to become non-convex. While linear models, with convex cost function, guarantee to find global minimum (convex optimization converges starting from any initial parameters).
  - Stochastic gradient descent applied to non-convex cost functions has no such convergence guarantee. And it is sensitive to the values of the initial parameters
  - For feedforward neural networks, it is important to initialize all weights to small random values.
  - The biases may be initialized to zero or to small positive values.
- Training feedforward neural networks – backpropagation training algorithm
  - For each training instance $x^{(i)}$, the algorithm does the following steps:
    - Forward pass: make a prediction (compute $\widehat{y^{(i)}} = f(x^{(i)})$)
    - Measure the error (compute $cost\left(\widehat{y^{(i)}}, y^{(i)}\right)$).
    - Backward pass: go through each layer in reverse to measure the error contribution from each connection.
    - Tweak the connection weights to reduce the error (update W and b).
- Output units
  - Linear
    - $\hat{y}_j = w_j^T h + b_j$
    - Minimize the mean squared error
  - Sigmoid
    - $\hat{y}_j = \sigma(w_j^T h + b_j)$
    - Minimize the cross-entropy
  - Softmax
    - $\hat{y}_j = softmax(w_j^T h + b_j)$
    - Minimize the cross-entropy
- Why replace step function with other activation functions in backpropagation algorithm?
  - Need to be differentiable
- Alternative activation functions:
  - Logistic function (sigmoid): $\sigma(z) = \frac{1}{1+e^{-z}}$
  - Hyperbolic tangent function: $tanh(z) = 2\sigma(2z) - 1$
  - Rectified linear units (ReLUs): $ReLU(z) = max(0, z)$
- Chain rule of calculus
  - $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$
- Backpropagation – forward pass
  - Calculate outputs given input patterns

- For each training instance
  - Feeds it to the network and computes the ouput of every neuron in each consecutive layer
  - Measures the network's output error (i.e., the difference between the true and the predicted output of the network)
  - Computes how much each neuron in the last hidden layer contributed to each output neuron's error
- Backpropagation – backward pass
  - Updates weights by calculating gradients
  - Measures how much of these error contributions came from each neuron in the previous hidden layer. Proceeds until the algorithm reaches the input layer
  - The last step is the gradient descent step on all the connection weights in the network, using the error gradients measured earlier

Module 7 Training deep feedforwards networks

- Challenges of training feedforward neural networks
  - Overfitting: risk of overfitting a model with large number of parameters
  - Vanishing/exploding gradients: hard to train lower layers
  - Training speed: slow training with large networks
- Overfitting
  - With large number of parameters, a network has a high degree
  - It can fit a huge variety of complex datasets
  - This flexibility also means that it is prone to overfitting on training set
- Avoid overfitting
  - Early stopping
  - l1 and l2 regularization
  - Max-norm regularization
  - Dropout
  - Data augmentation
- Early stopping
  - As the training steps go by, its prediction error on the training/validation set naturally goes down
  - After a while the validation error stops decreasing and starts to go back up
    - The model has started to overfit the training data
  - In the early stopping, we stop training when the validation error reaches a minimum
- l1 and l2 regularization
  - Penalize large values of weights w_j
    - $\tilde{J}(w) = J(w) + \lambda R(w)$
  - Two questions
    - How should we define $R(w)$
    - How do we determine λ
  - l1 regression: $R(w) = \lambda \sum_{i=1}^{n} |w_i|$ is added to the cost function
    - $\tilde{J}(w) = J(w) + \lambda \sum_{i=1}^{n} |w_i|$
  - l2 regression: $R(w) = \lambda \sum_{i=1}^{n} w_i^2$ is added to the cost function
    - $\tilde{J}(w) = J(w) + \lambda \sum_{i=1}^{n} w_i^2$
- Max-norm regularization
  - Constrains the weights w_j of the incoming connections for each neuron j
    - Prevents them from getting too large
  - After each training step, clip w_j as below, if $||w_j||_2 > r$

$$w_j \leftarrow w_j \frac{r}{||w_j||_2}$$

    - r is the max-norm hyperparameter
    - $||w_j||_2 = \left(\sum_i w_{i,j}^2\right)^{\frac{1}{2}} = \sqrt{w_{1,j}^2 + w_{2,j}^2 + \cdots + w_{n,j}^2}$
- Dropout
  - At each training step, each neuron drops out temporarily with a probability p
    - The hyperparameter p is called the dropout rate
    - A neuron will be entirely ignored during this training step

        - It may be active during the next step
        - Exclude the output neurons
    - After training, neurons don't get dropped anymore
- Data augmentation
    - One way to make a model generalize better is to train it on more data
    - This will reduce overfitting
    - Create fake data and add it to the training set
        - Shift, rotate and resize an image and add the resulting pictures to the training set
- Vanishing/exploding gradients problem
    - Vanishing gradients problem: the backpropagation goes from output to input layer, and propagates the error gradient on the way. Gradients often get smaller and smaller as the algorithm progresses down to the lower layers. As a result, the gradient descent update leaves the lower layer connection weights virtually unchanged
- Overcoming the vanishing gradient
    - Parameter initialization strategies
    - Nonsaturating activation function
    - Batch normalization
    - Gradient clipping
- Parameter initialization strategies
    - Non-linearity of the network → cost function becomes non-convex
    - Stochastic gradient descent on non-convex cost functions performs is sensitive to the to the values of the initial parameters
    - Initial parameters need to break symmetry between different units (break symmetry: two hidden units with the same activation function connected to the same inputs must have different initial parameters to compute a different function. The goal of having each unit compute a different function. Otherwise, the output will be the same and training will serve no purpose)
    - It motivates random initialization of the parameters
        - Typically, we set the biases to constants, and initialize only the weights randomly.
    - We need the signals to flow properly in both directions.
    - The Glorot and Bengio initialization proposed that:
        - The variance of the outputs of each layer to be equal to the variance of its inputs
        - The gradients to have equal variance before and after flowing through a layer in the reverse direction
    - It is not possible to guarantee both unless each layer has an equal number of inputs and neurons
    - Based on the Xavier initialization, the weights are initialized using normal distribution with mean 0 and the following standard deviation
    - fan_{in} and fan_{out} are the number of inputs and neurons for the layer whose weights are being initialized
    - $fan_{avg} = \frac{2}{fan_{in}+fan_{out}}$
    - Glorot initialization, for none, logistic, sigmoid, and tanh: $\sigma^2 = \frac{1}{fan_{avg}}$

- He initialization, for ReLU: $\sigma^2 = \frac{1}{fan_{in}}$

- Nonsaturating activation function
  - Dying ReLUs problem
    - During training, some neurons stop outputting anything other than 0, e.g., when the weighted sum of the neuron's inputs is negative, it starts outputting 0
  - Use leaky ReLU instead: $LeakyReLU_\alpha(z) = max(\alpha z, z)$
    - α is the slope of the function for z < 0
  - Randomized Leaky ReLU (RReLU)
    - α is picked randomly during training, and it is fixed during testing
  - Parametric Leaky ReLU (PReLU)
    - Learn α during training (instead of being a hyperparameter)
  - Exponential Linear Unit (ELU)
    - $ELU_\alpha = \begin{cases} \alpha(exp(z) - 1) & if\ z < 0 \\ z & if\ z \geq 0 \end{cases}$
  - In general, we choose activation function
    - logistic < tanh < ReLU < leaky ReLU (and its variants) < ELU
    - If you care about runtime performance, then leaky ReLUs works better than ELUs
- Batch normalization
  - Batch normalization makes the learning of layers in the network more independent of each other
    - It is a technique to address the problem that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change
  - The technique consists of adding an operation in the model just before the activation function of each layer
  - It's zero-centering and normalizing the inputs, then scaling and shifting the result
    - Estimates the inputs' mean and standard deviation of the current mini-batch

$$\mu_B = \frac{1}{m_B}\sum_{i=1}^{m_B} x^{(i)}$$

$$\sigma_B^2 = \frac{1}{m_B}\sum_{i=1}^{m_B}\left(x^{(i)} - \mu_B\right)^2$$

$$\widehat{x^{(i)}} = \frac{x^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$z^{(i)} = \gamma\widehat{x^{(i)}} + \beta$$

- Gradient clipping
  - clip the gradients during backpropagation so that they never exceed some threshold
  - Setting the clipvalue or clipnorm argument when creating an optimizer
  - clipvalue=1.0 and clipnorm=1.0: values between -1.0 and 1.0
  - clipvalue=1.0: [0.9, 100.0] ⇒ [0.9, 1.0]
  - clipnorm=1.0: [0.9, 100.0] ⇒ [0.00899964, 0.9999595]
- Optimization algorithms

- Momentum
- Nesterov momentum
- AdaGrad
- RMSProp
- Adam optimization

- Momentum
  - It measures the resistance to change in motion. The higher momentum an object has, the harder it is to stop it
  - It cares about what previous gradients were
  - At each iteration, it adds the local gradient to the momentum vector m

$$m_i = \beta m_i + \eta \frac{\partial J(w)}{\partial w_i}$$

$$w_i^{(next)} = w_i - m$$

  - β is called momentum, and it is between 0 and 1
- Nesterov momentum
  - A small variant to momentum optimization
  - Faster
  - ∇1 represents the gradient of the cost function measured at the starting point w, and ∇2 represents the gradient at the point located at $w + \beta m$
  - Measure the gradient of the cost function slightly ahead in the direction of the momentum (not at the local position)

$$m_i = \beta m_i + \eta \frac{\partial J(w + \beta m)}{\partial w_i}$$

$$w_i^{(next)} = w_i - m_i$$

- AdaGrad
  - It keeps track of a learning rate for each parameter
  - Adapts the learning rate over time (adaptive learning rate)
  - Decays the learning rate faster for steep dimensions than for dimensions with gentler slopes
  - For each feature w_i, we do the following steps:

$$s_i = s_i + \left(\frac{\partial J(w)}{\partial w_i}\right)^2$$

$$w_i^{(next)} = w_i - \frac{\eta}{\sqrt{s_i + \epsilon}} \frac{\partial J(w)}{\partial w_i}$$

- RMSProp
  - AdaGrad often stops too early when training neural networks: The learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum
  - The RMSProp fixed the AdaGrad problem.
  - It is like the AdaGrad problem, but accumulates only the gradients from the most recent iterations (not from the beginning of training)
  - For each feature w_i, we do the following steps:

$$s_i = \beta s_i + (1-\beta)\left(\frac{\partial J(w)}{\partial w_i}\right)^2$$

$$w_i^{(next)} = w_i - \frac{\eta}{\sqrt{s_i+\epsilon}}\frac{\partial J(w)}{\partial w_i}$$

- Adam optimization
  - Adam (Adaptive moment estimation) combines the ideas of Momentum optimization and RMSProp
  - Like Momentum optimization, it keeps track of an exponentially decaying average of past gradients
  - Like RMSProp, it keeps track of an exponentially decaying average of past squared gradients

1. $\mathbf{m}^{(\text{next})} = \beta_1 \mathbf{m} + (1-\beta_1)\nabla_{\mathbf{w}} J(\mathbf{w})$
2. $\mathbf{s}^{(\text{next})} = \beta_2 \mathbf{s} + (1-\beta_2)\nabla_{\mathbf{w}} J(\mathbf{w}) \otimes \nabla_{\mathbf{w}} J(\mathbf{w})$
3. $\mathbf{m}^{(\text{next})} = \frac{\mathbf{m}}{1-\beta_1^T}$
4. $\mathbf{s}^{(\text{next})} = \frac{\mathbf{s}}{1-\beta_2^T}$
5. $\mathbf{w}^{(\text{next})} = \mathbf{w} - \eta \mathbf{m} \oslash \sqrt{\mathbf{s}+\epsilon}$

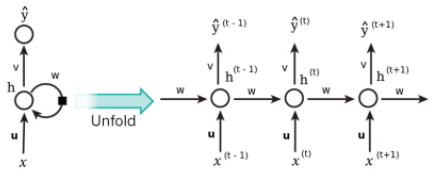
- $\otimes$ and $\oslash$ represent the element-wise multiplication and division.
- Steps 1, 2, and 5: similar to both Momentum optimization and RMSProp.
- Steps 3 and 4: since $\mathbf{m}$ and $\mathbf{s}$ are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost $\mathbf{m}$ and $\mathbf{s}$ at the beginning of training.

Module 8 Convolutional neural network

- Convolutional neural networks (CNN) can tackle the deep neural networks challenges
  - CNN is a type of neural network that can take advantage of shape information
  - It applies a series of filters to the raw pixel data of an image to extract and learn higher-level features, which the model can then use for classification
- Receptive fields and filters
  - Imagine a flashlight that is shining over the top left of the image
  - The region that it is shining over is called the receptive field
  - This flashlight is called a filter
  - A filter is a set of weights
  - A filter is a feature detector, e.g., straight edges, simple colors, and curves
- Convolution operation
  - Convolution takes a filter and multiplying it over the entire area of an input image
  - Imagine this flashlight (filter) sliding across all the areas of the input image
- CNN components
  - Convolutional layers: apply a specified number of convolution filters to the image
  - Pooling layers: downsample the image data extracted by the convolutional layers to reduce the dimensionality of the feature map in order to decrease processing time
  - Dense layers: a fully connected layer that performs classification on the features extracted by the convolutional layers and downsampled by the pooling layers
  - A CNN is composed of a stack of convolutional modules
  - Each module consists of a convolutional layer followed by a pooling layer
  - The last module is followed by one or more dense layers that perform classification
  - The final dense layer contains a single node for each target class in the model, with a softmax activation function
- Convolutional layer
  - Sparse interactions
  - Each neuron in the convolutional layers is only connected to pixels in its receptive field (not every single pixel)
  - Each neuron applies filters on its receptive field.
    - Calculates a weighted sum of the input pixels in the receptive field
  - Adds a bias, and feeds the result through its activation function to the next layer
  - The output of this layer is a feature map (activation map)
  - Parameter sharing
  - All neurons of a convolutional layer reuse the same weights
  - They apply the same filter in different positions
  - Whereas in a fully-connected network, each neuron had its own set of weights
- Padding
  - Zero padding: in order for a layer to have the same height and width as the previous layer, it is common to add zeros around the inputs
  - In TensorFlow, padding can be either SAME or VALID to have zero padding or not.
- Stride
  - It is the distance between two consecutive receptive fields
  - The stride controls how the filter convolves around the input volume

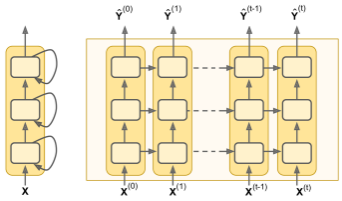
- Stacking multiple features maps
  - Each convolutional layer can be composed of several feature maps of equal sizes
  - A convolutional layer simultaneously applies multiple filters to its inputs
- Activation function
  - After calculating a weighted sum of the input pixels in the receptive fields, and adding biases, each neuron feeds the result through its ReLU activation function to the next layer
  - The purpose of this activation function is to add non linearity to the system
- Pooling layer
  - After the activation functions, we can apply a pooling layer
  - Its goal is to subsample (shrink) the input image
    - To reduce the computational load, the memory usage, and the number of parameters
  - Each neuron in a pooling layer is connected to the outputs of a receptive field in the previous layer
  - A pooling neuron has no weights
  - It aggregates the inputs using an aggregation function such as the max or mean
- Fully connected layer
  - This layer takes an input from the last convolution module, and outputs an N dimensional vector
    - N is the number of classes that the model has to choose from
  - Each number in this N dimensional vector represents the probability of a certain class
- Flattening
  - We need to convert the output of the convolutional part of the CNN into a 1D feature vector
  - It gets the output of the convolutional layers, flattens all its structure to create a single long feature vector to be used by the dense layer for the final classification

Module 9 Recurrent neural networks

- N-gram language model
  - A n-gram is a chunk of n consecutive words
  - Collect statistics about how frequent different n-grams are, and use these to predict next word.
- Problems with n-gram language model
  - Sparsity: Increasing n makes sparsity problems worse
    - Typically we can't have n bigger than 5
  - Storage: For "students opened their w_j", we need to store count for all possible 4-grams
    - The model size is in the order of O(exp(n))
    - Increasing n makes model size huge
- MLP model
  - Input: words $x^{(1)}, x^{(2)}, x^{(3)}, x^{(4)}$
  - Input layer: one-hot vectors $e^{(1)}, e^{(2)}, e^{(3)}, e^{(4)}$
  - Hidden layer: $h = f(w^T e)$, f is an activation function
  - Output: $\hat{y} = softmax(v^T h)$
  - Improvements over n-gram LM:
    - No sparsity problem
    - Model size is O(n) not O(exp(n))
  - Remaining problems:
    - It is fixed 4 in our example, which is small
    - We need a neural architecture that can process any length input
- Recurrent neural networks
  - The idea behind Recurrent neural networks (RNN) is to make use of sequential data
    - Until here, we assume that all inputs (and outputs) are independent of each other
    - Independent input (output) is a bad idea for many tasks, e.g., predicting the next word in a sentence (it's better to know which words came before it).
  - They can analyze time series data and predict the future
  - They can work on sequences of arbitrary lengths, rather than on fixed-sized inputs
  - Neurons in an RNN have connections pointing backward
  - RNNs have memory, which captures information about what has been calculated so far
  - Unfolding the network: represent a network against the time axis
    - We write out the network for the complete sequence
    - 

  - $h^{(t)} = f\left(u^T x^{(t)} + w h^{(t-1)}\right)$, where f is an activation function
  - $\hat{y}^{(t)} = g\left(v h^{(t)}\right)$, where g can be the softmax function
  - $cost\left(y^{(t)}, \widehat{y^{(t)}}\right) = cross\_entropy\left(y^{(t)}, \widehat{y^{(t)}}\right) = -\sum y^{(t)} log \widehat{y^{(t)}}$
- Recurrent neurons – weights
  - Each recurrent neuron has three sets of weights: u, w, and v

- u: the weights for the inputs $x^{(t)}$
- w: the weights for the hidden state of the previous time step $h^{(t-1)}$
- $h^{(t)}$: is the hidden state (memory) at time step t
  - $h^{(t)} = f\left(u^T x^{(t)} + w h^{(t-1)}\right)$
  - $h^{(0)}$ is the initial hidden state
- v: the weights for the hidden state of the current time step $h^{(t)}$

- Deep rnn
  - Stacking multiple layers of cells
  - 
- RNN design pattern
  - Sequence to vector
    - takes a sequence of inputs, and ignore all outputs except for the last one
    - E.g., you could feed the network a sequence of words corresponding to a movie review, and the network would output a sentiment score
  - vector to sequence
    - takes a single input at the first time step, and let it output a sequence
    - E.g., the input could be an image, and the output could be a caption for that image
  - Sequence to sequence
    - takes a sequence of inputs and produce a sequence of outputs
    - Useful for predicting time series such as stock prices: you feed it the prices over the last N days, and it must output the prices shifted by one day into the future
    - Here, both input sequences and output sequences have the same length
  - Encoder-decoder
    - a sequence-to-vector network (encoder), followed by a vector-to-sequence network (decoder)
    - E.g., translating a sentence from one language to another
- Training RNNs
  - we should unroll it through time and then simply use regular backpropagation
  - This strategy is called backpropagation through time (BPTT).
- Backpropagation through time
  - Forward pass through the unrolled network (represented by the dashed arrows)
  - The cost function is $c\left(\widehat{y^{tmin}}, \widehat{y^{tmin+1}}, \dots, \widehat{y^{tmax}}\right)$, where tmin and tmax are the first and last output time steps, not counting the ignored outputs
  - Propagate backward the gradients of that cost function through the unrolled network (represented by the solid arrows)
  - The model parameters are updated using the gradients computed during BPTT
  - The gradients flow backward through all the outputs used by the cost function, not just through the final output
- RNN problems
  - Sometimes we only need to look at recent information to perform the present task

- In such cases, where the gap between the relevant information and the place that it's needed is small, RNNs can learn to use the past information
- But, as that gap grows, RNNs become unable to learn to connect the information
- RNNs may suffer from the vanishing/exploding gradients problem
- To solve these problem, long short-term memory (LSTM) have been introduced
- In LSTM, the network can learn what to store and what to throw away

- LSTM
  - An LSTM contains four interacting layers in each cell
  - In LSTM state is split in two vectors
    - $h^{(t)}$ (h stands for hidden): the short-term state
    - $c^{(t)}$ (c stands for cell): the long-term state
  - The cell state (long-term state), the horizontal line on the top of the diagram
  - The LSTM can remove/add information to the cell state, regulated by three gates
    - Forget gate, input gate and output gate
- Step-by-step LSTM walk through
  - decides what information we are going to throw away from the cell state
    - This decision is made by a sigmoid layer, called the forget gate layer
    - It looks at $h^{(t-1)}$ and $x^{(t)}$, and outputs a number between 0 and 1 for each number in the cell state $c^{(t-1)}$
    - 1 represents completely keep this, and 0 represents completely get rid of this
    - $f^{(t)} = \sigma\left(u_f^T x^{(t)} + w_f h^{(t-1)}\right)$
  - decides what new information we are going to store in the cell state
    - A sigmoid layer, called the input gate layer, decides which values we will update
    - A tanh layer creates a vector of new candidate values that could be added to the state
    - $i^{(t)} = \sigma\left(u_i^T x^{(t)} + w_i h^{(t-1)}\right)$
    - $\widetilde{c^{(t)}} = tanh\left(u_{\tilde{c}}^T x^{(t)} + w_{\tilde{c}} h^{(t-1)}\right)$
  - updates the old cell state $c^{(t-1)}$, into the new cell state $c^{(t)}$
    - We multiply the old state by $f^{(t)}$, forgetting the things we decided to forget earlier
    - Then we add it $i^{(t)} \otimes \tilde{c}^{(t)}$
    - This is the new candidate values, scaled by how much we decided to update each state value
    - $c^{(t)} = f^{(t)} \otimes c^{(t-1)} + i^{(t)} \otimes \widetilde{c^{(t)}}$
  - decides about the output
    - First, runs a sigmoid layer that decides what parts of the cell state we are going to output
    - Then, puts the cell state through tanh and multiplies it by the output of the sigmoid gate (output gate), so that it only outputs the parts it decided to
    - $o^{(t)} = \sigma\left(u_o^T x^{(t)} + w_o h^{(t-1)}\right)$
    - $h^{(t)} = o^{(t)} \otimes tanh\left(c^{(t)}\right)$
- Gated recurrent unit (GRU)
  - The GRU cell is a simplified version of the LSTM cell
  - Instead of separately deciding what to forget and what to add to the new information to, it makes those decisions together

- It only forgets when it is going to input something in its place
- It only inputs new values to the state when it forgets something older
- $c^{(t)} = f^{(t)} \otimes c^{(t-1)} + (1 - f^{(t)}) \otimes \widetilde{c^{(t)}}$

Module 10 Transformers and attention

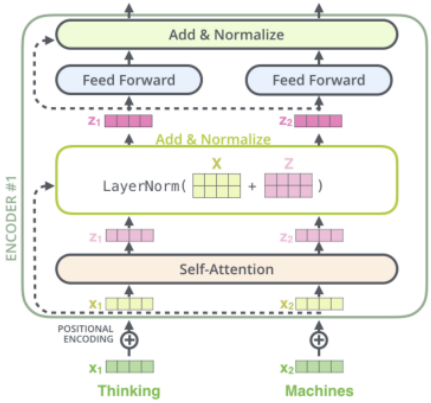
- Word embeddings
  - Numerical operations on the words can get semantic match
    - E.g., king + woman – man ≈ queen
  - Problems: Word embeddings are context-free
    - E.g., walk can be either noun or verb
  - Solution: Create contextualized representation
- Problems with RNNs – motivation for transformers
  - Sequential computations prevents parallelization
  - Despite GRUs and LSTMs, RNNs still need attention mechanisms to deal with long range dependencies
  - Attention gives us access to any state…Maybe we don't need the costly recursion?
  - Then NLP can have deep models, solves our computer vision envy!
- Attention preliminaries
  - Mimics the retrieval of a value *vi* for a query *q* based on a key *ki* in a database, but in a probabilistic fashion
- Dot-product attention
  - Queries, keys and values are vectors
  - Output is a weighted sum of the values
  - Weights are computed as the scaled dot-product (similarity) between the query and the keys

$$Attention(q, K, V) = \sum_i Similarity(q, k_i) \cdot v_i = \sum_i \frac{e^{q \cdot k_i / \sqrt{d_k}}}{\sum_j e^{q \cdot k_j / \sqrt{d_k}}} v_i$$

  - Can stack multiple queries into a matrix Q

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

  - Self-attention: Let the word embeddings be the queries, keys and values, i.e., let the words select each other
- Full encoder block
  - It consist of
    - Multi-headed self-attention
    - Feedforward NN (FC 2 layers)
    - Skip connections
    - Layer normalization – similar to batch normalization but computed over features (words/tokens) for a single sample

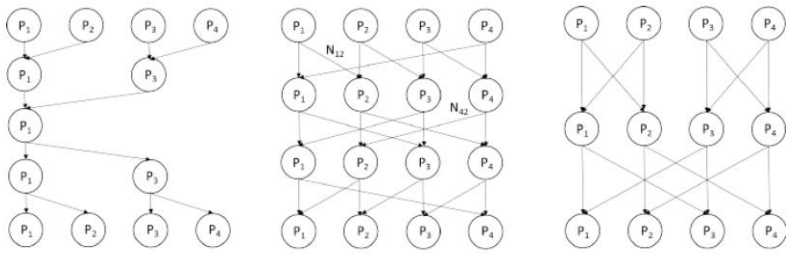


- Positional encodings
  - Encoder block consisting of
    - Attention mechanism has no locality bias – no notion of word order
    - Add positional encodings to input embeddings o let model learn relative positioning
    - $PE(pos, 2i) = sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$
    - $PE(pos, 2i+1) = cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$
- Producing the output text
  - Encoder block consisting of
    - The output from the decoder is passed through a final fully connected linear layer with a softmax activation function
    - Produces a probability distribution over the pre-defined vocabulary of output words (tokens)
    - Greedy decoding picks the word with the highest probability at each time step
- Bert
  - Bidirectional Encoder Representations from Transformers
    - Self-supervised pre-training of Transformers encoder for language understanding
    - Fine-tuning for specific downstream task
- Distillation
  - To apply huge and computationally expensive pre-trained language models to low-latency applications
    - Train SOTA teacher model (pre-training + fine-tuning)
    - Train smaller student model that mimics the teacher's output on a large dataset on unlabeled data

Module 11 Distributed deep learning

- Training deep neural networks is
  - Computationally intensive
  - Time consuming
  - Because
    - Massive amount of training dataset
    - Large number of parameters
- Descent
  - Gradient descent: $w = w - \eta \nabla J(w)$
  - Stochastic gradient descent: $w = w - \eta \tilde{g} J(w)$
    - $\tilde{g}$: gradient at a randomly chosen point
  - Mini-batch gradient descent: $w = w - \eta \widetilde{g_B} J(w)$
    - $\tilde{g}$: gradient with respect to a set of B randomly chosen points
- Scalable training
  - Data parallelism
  - Model parallelism
- Data parallelism
  - Replicate a whole model on every device
  - Train all replicas simultaneously, using a different mini-batch for each
  - k devices
  - $J_j(w) = \sum_{i=1}^{b_j} l(y_i, \hat{y_i}), \forall j = 1,2,\dots,k$
  - $\widetilde{g_B} J_j(w)$: gradient of $J_j(w)$ with respect to a set of B randomly chosen points at device j
  - Compute $\widetilde{g_B} J_j(w)$ on each device j
  - Compute the mean of the gradients
  - $\widetilde{g_B} J(w) = \frac{1}{k} \sum_{j=1}^{k} \widetilde{g_B} J_j(w)$
  - Update the model
  - $w = w - \eta \widetilde{g_B} J(w)$
- Data parallelization design issues
  - The aggregation algorithm
  - Communication synchronization and frequency
  - Communication compression
- Aggregation algorithm
  - How to aggregate gradients (compute the mean of the gradients)
  - Centralized – parameter server
  - Decentralized – all-reduce
  - Decentralized – gossip
- Parameter server
  - Store the model parameters outside of the workers
  - Workers periodically report their computed parameters or parameter updates to a (set of) parameter server(s) (PSs)
- All-reduce
  - Mirror all the model parameters across all workers (no PS)

- Workers exchange parameter updates directly via an all-reduce operation
- Gossip
  - No PS, and no global model
  - Every worker communicates updates with their neighbors.
  - The consistency of parameters across all workers only at the end of the algorithm
- Reduce and all-reduce
  - Reduce: reducing a set of numbers into a smaller set of numbers via a function
  - E.g., sum([1, 2, 3, 4, 5]) = 15
  - Reduce takes an array of input elements on each process and returns an array of output elements to the root process.
  - All-Reduce stores reduced results across all processes rather than the root process
- All-reduce implementation
  - All-to-all all-reduce
  - Master-worker all-reduce
  - Tree all-reduce
  - Round-robin all-reduce
  - Butterfly all-reduce
  - Ring all-reduce
- All-to-all all-reduce
  - Send the array of data to each other
  - Apply the reduction operation on each process
  - Too many unnecessary messages
- Master-worker all-reduce
  - Selecting one process as a master, gather all arrays into the master
  - Perform reduction operations locally in the master
  - Distribute the result tot eh other processes
  - The master becomes a bottleneck (not scalable)
- Other implementation
  - Some try to minimize bandwidth
  - Some try to minimize latency

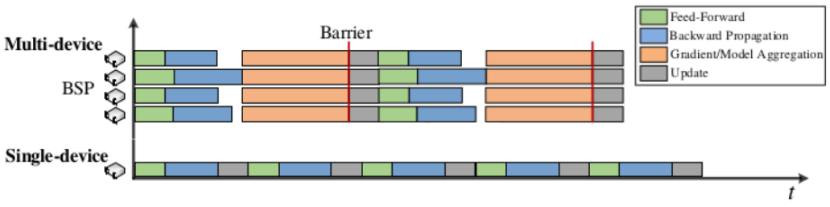


- Ring all-reduce
  - Two phases
    - First, the share-reduce phase
    - Then, the share-only phase
  - In the share-reduce phase, each process p sends data to the process (p+1)%m
    - m is the number of processes, and % is the modulo operator

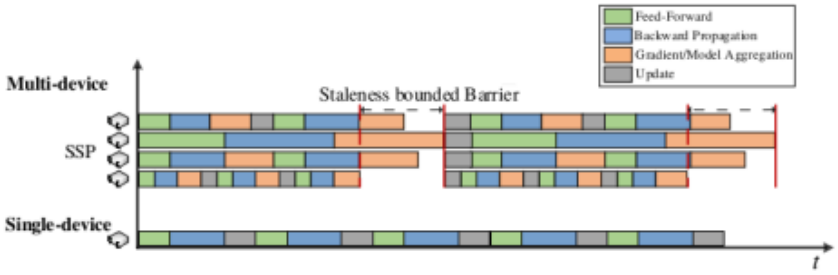
- The array of data on each process is divided to m chunks (m=4 here)
- Each one of these chunks will be indexed by i going forward
- When each process receives the data from the previous process, it applies the reduce operator (e.g., sum)
  - The reduce operator should be associative and commutative
- It then proceeds to send it to the next process in the ring
- The share-reduce phase finishes when each process holds the complete reduction of chunk i.
- At this point each process holds a part of the end result
- The share-only step is the same process of sharing the data in a ring-like fashion without applying the reduce operation
- This consolidates the result of each chunk in every process

- Master-worker all-reduce vs. ring all-reduce
  - N: number of elements, m: number of processes
  - Master-worker all-reduce
    - First each process sends N elements to the master: $N \times (m - 1)$ messages
    - Then the master sends the results back to the process: another $N \times (m - 1)$ messages
    - Total network traffic is $2(N \times (m - 1))$, which is proportional to m
  - Ring all-reduce
    - In the share-reduce step each process send N/m messages, and it does it m-1 times: N/m*(m-1) messages
    - On the share-only step, each process sends the result for the chunk it calculated: another N/m*(m-1) messages
    - Total network traffic is 2(N/m*(m-1))
- Communication Synchronization
  - Synchronizing the model replicas in data-parallel training requires communication
    - Between workers, in all-reduce
    - Between workers and parameter servers, in centralized architecture
  - The communication synchronization decides how frequently all local models are synchronized with others
  - It will influence
    - The communication traffic
    - The performance
    - The convergence of model training
  - There is a trade-off between the communication traffic and the convergence
- Reducing synchronization overhead
  - To relax the synchronization among all workers
  - The frequency of communication can be reduced by more computation in one iteration
- Communication synchronization models
  - Synchronous
  - Stale-synchronous
  - Asynchronous
  - Local SGD
- Synchronous

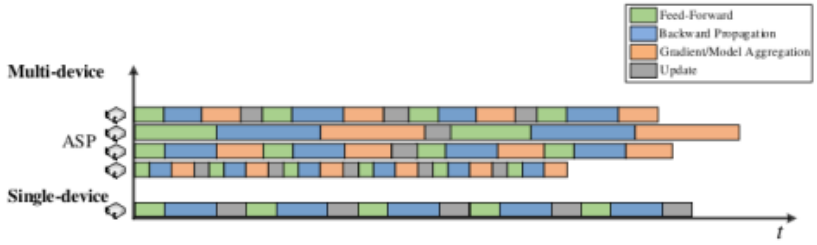
- After each iteration, the workers synchronize their parameter updates
- Every worker must wait for all workers to finish the transmission of all parameters in the current iteration, before the next training
- Stragglers can influence the overall system throughput
- High communication cost that limits the system scalability



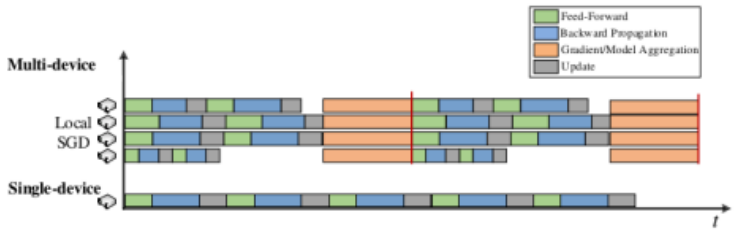
- Stale synchronous
  - For a maximum staleness bound s, the update formula of worker i at iteration t+1
  - $w_{i,t+1} = w_0 - \eta\left(\sum_{k=1}^{t}\sum_{j=1}^{n} G_{j,k} + \sum_{k=t-s}^{t} G_{i,k} + \sum_{(j,k)\in S_{i,t+1}} G_{j,k}\right)$
  - The update has three parts
    - Guaranteed pre-window updates from clock 1 to t over all workers
    - Guaranteed read-my-writes in-window updates made by the querying worker i
    - Best-effort in-window updates. $S_{i,t+1}$ is some subset of the updates from other workers during period [t − s]

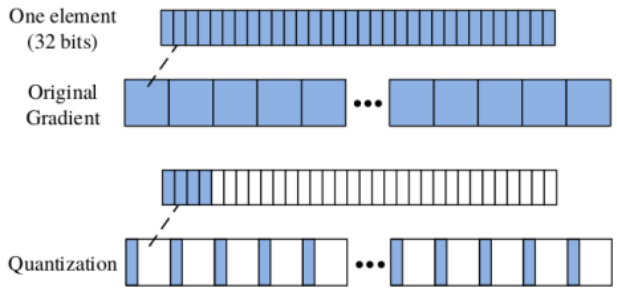


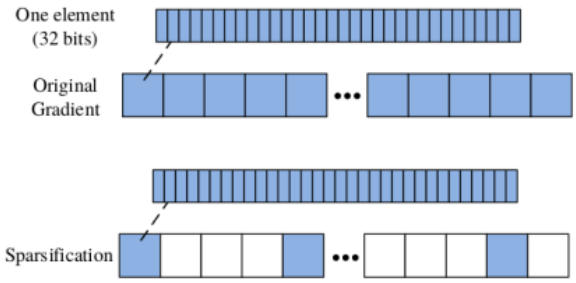
- Asynchronous
  - It completely eliminates the synchronization
  - Each work transmits its gradients to the PS after it calculates the gradients
  - The PS updates the global model without waiting for the other workers
  - $w_{t+1} = w_t - \eta\sum_{i=1}^{n} G_{i,t-\tau_{k,i}}$
  - $\tau_{k,i}$ is the time delay between the moment when worker i calculates the gradient at the current iteration



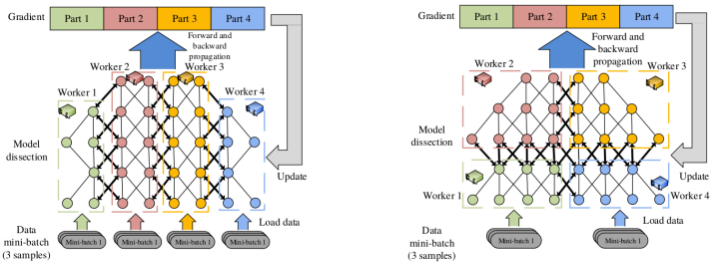
- Local SGD
  - All workers run several iterations, and then averages all local models into the newest global model
  - If I_t represents the synchronization timestamps, then:

- $w_{i,t+1} = \begin{cases} w_{i,t} - \eta G_{i,t} & if\ t+1 \notin I_t \\ w_{i,t} - \eta \frac{1}{n} \sum_{i=1}^{n} G_{i,t} & if\ t+1 \in I_t \end{cases}$
- 

- Communication compression
  - Reduce the communication traffic with little impact on the model convergence
  - Compress the exchanged gradients or models before transmitting across the network
  - Quantization
  - Sparsification
- Quantization
  - Useing lower bits to represent the data
  - The gradients are of low precision
  - 

- Sparsification
  - Reducing the number of elements that are transmitted at each iteration
  - Only significant gradients are required to update the model parameter to guarantee the convergence of the training
  - E.g., the zero-valued elements are no need to transmit
  - 

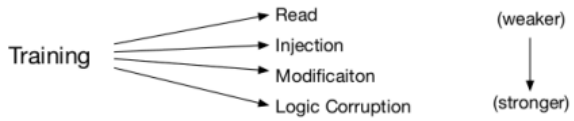
- Model parallelization
  - The model is split across multiple devices
  - Depends on the architecture of the NN

- 

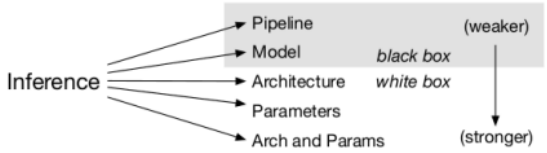
- Model parallelization – hash partitioning
  - Randomly assign vertices to devices proportionally to the capacity of the devices by using a hash function
- Model parallelization – critical path
  - Assigning the complete critical path to the fastest device
  - Critical path: the path with the longest computation time from source to sink vertex
- Model parallelization – multi-objective heuristics
  - Different objectives, e.g., memory, importance, traffic, and execution time
- Model parallelization – reinforcement learning
  - $J(w) = E_{p\sim\pi(p|G,w)}[R(P)|G]$
  - Objective: $argmin_w J(w)$
  - G: input neural graph
  - R: runtime
  - J(w): expected runtime
  - w: trainable parameters of policy
  - $\pi(P|G,w)$: policy
  - P: output placements $\in$ {1, 2, ..., num ops}^num devices
  - RL reward function based on execution runtime
  - The RL policy is defined as a seq-to-seq model
  - RNN Encoder receives graph embedding for each operation
  - RNN Decoder predicts a device placement for each operation
  - Grouping operations
  - Prediction is for group placement, not for a single operation

Module 13 Distributed roubust learning

- Adversary goal
  - Confidentiality and privacy: confidentiality of the model or the data
  - Integrity: integrity of the predictions
  - Availability: availability of the system deploying machine learning
- Adversarial capabilities for integrity attacks
  - Training phase

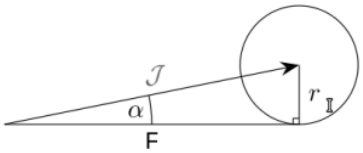


    -
  - Inference phase
    - White box
    - Black box



    -
- Our focus and goal
  - Data parallelization
  - Each worker is prone to adversarial attack
  - Adversarial attacks: some unknown subset of computing devices are compromised and behave adversarially (e.g., sending out malicious messages)
  - Our goal: integrity of the model in the training phase
- Distributed stochastic gradient descent
  - One parameter server, and n workers
  - Computation is divided into synchronous rounds
  - During round *t*, the parameter server broadcasts its parameter vector w $\in R^d$ to all the workers
  - At each round *t*, each correct worker *i* computes $G_i(w_t, \beta)$
  - $G_i(w_t, \beta)$: the local estimate of the gradient of the loss function $\nabla J(w_t)$
  - β: a mini-batch of i.i.d. (independent and identically distributed) samples drawn from the dataset
  - $G_i(w_t, \beta) = \frac{1}{|\beta|}\sum_{x\in\beta} \nabla l_i(w_t, x)$
  - The parameter server computes F(G_1, G_2, …, G_n)
  - Gradient Aggregation Rule (GAR): $F(G_1, G_2, \dots, G_n) = \frac{1}{n}\sum_{i=1}^{n} G_i$
  - The parameter server updates the parameter vector $w \leftarrow w - \gamma F(G_1, G_2, \dots, G_n)$
- Distributed SGD with Byzantine workers
  - Among the n workers, f of them are possibly Byzantine (behaving arbitrarily)
  - A Byzantine worker b proposes a vector G_b that can deviate arbitrarily from the vector it is supposed
- Averaging GAR and Byzantine workers

- Averaging GAR: $F(\mathrm{G}_1, \mathrm{G}_2, \ldots, \mathrm{G}_\mathrm{n}) = \frac{1}{\mathrm{n}} \sum_{\mathrm{i}=1}^{\mathrm{n}} \mathrm{G}_\mathrm{i}$
- $w \leftarrow w - \gamma F(G_1, G_2, \ldots, G_n)$
- Even a single Byzantine worker can prevent convergence
- Proof: if the Byzantine worker proposes $G_n = nU - \sum_{i=1}^{n-1} G_i$, then F=U

- (α, f)-Byzantine-resilience
  - Assume n workers, where f of them are Byzantine workers
  - $\alpha \in [0, \pi/2]$ and $f \in \{0, \ldots, n\}$
  - $\left(G_1, \ldots, G_{n-f}\right) \in (R^d)^{n-f}$ are i.i.d. (independent and identically distributed) random vectors
    - G_i ~ g
    - E[g]= $\mathscr{J}$, where $\mathscr{J} = \nabla J(w)$
  - $\left(B_1, \ldots, B_f\right) \in (R^d)^f$ are random vectors, possibly dependent between them and the vectors $\left(G_1, \ldots, G_{n-f}\right)$
  - A GAR F is said to be (α, f)-Byzantine-resilience if, for any 1 ≤ j_1 < · · · < j_f ≤ n, the vector F(G_1, · · · , B_1, · · · , B_f, · · · , G_n) satisfies
    - Vector F that is not too far from the real gradient $\mathscr{J}$, i.e., ||E[F] − $\mathscr{J}$|| ≤ r
    - Moments of F should be controlled by the moments of the (correct) gradient estimator g, where E[g] = $\mathscr{J}$
    - 

- Byzantine-resilience GAR
  - Median
  - Krum
  - Multi-Krum
  - Brute
- Median
  - n≥2f+1
  - median$(x_1, \ldots, x_n)$=arg $min_{x \in R} \sum_{i=1}^{n} |x_i - x|$
  - d: the gradient vectors dimension
  - Geometric median: $F = GeoMed(G_1, \ldots, G_n) = argmin_{G \in R^d} \sum_{i=1}^{n} \|G_i - G\|$
  - Marginal median:

$$F = MarMed(G_1, \ldots, G_n) = \begin{pmatrix} mdian(G_1[1], \ldots, G_n[1]) \\ \ldots \\ median(G_1[d], \ldots, G_n[d]) \end{pmatrix}$$

- Krum
  - n≥2f+3
  - Idea: to preclude the vectors that are too far away
  - $s(i) = \sum_{i \to j} \left\| G_i - G_j \right\|^2$, the score of the worker i
  - $i \to j$ denotes that G_j belongs to the n-f-2 closest vectors to G_i
  - $F(\mathrm{G}_1, \mathrm{G}_2, \ldots, \mathrm{G}_\mathrm{n}) = G_{i_*}$

- G_{i_*} refers to the worker minimizing the score, $s(i_*) \leq s(i)$ for all i

- Multi-Krum
  - It computes the score for each vector proposed (as in Krum)
  - It selects m vectores $G_{1_*}, \ldots, G_{m_*}$, which score the best ($1 \leq m \leq n - f - 2$)
  - It outputs their average $\frac{1}{m}\sum_i G_{i_*}$
  - The cases m = 1 and m = n correspond to Krum and averaging, respectively
- Brute
  - $n \geq 2f + 1$
  - $Q = G_1, \ldots, G_n$
  - $R = X | X \subset Q, |X| = n - f$
    - The set of all the subsets of n - f
  - $S = argmin_{X \in R}\left(max_{(G_i, G_j) \in X^2}(\|G_i - G_j\|)\right)$
    - Selects the n – f most clumped gradients among the submitted ones
  - $F(G_1, G_2, \ldots, G_n) = \frac{1}{n-f}\sum_{G \in S} G$
- Weak Byzantine resilience
  - Limitation of previous aggregation methods
  - If gradient dimension d >> 1, then the distance function between two vectors ||X−Y||_p, cannot distinguish these two cases:
    - Does X and Y disagree a bit on each coordinate?
    - Does X and Y disagree a lot on only one?
- Strong byzantine resilience
  - Ensuring convergence (as in weak Byzantine resilience functions)
  - Ensures that each coordinate is agreed on by a majority of vectors that were selected by a Byzantine resilient aggregation rule A
  - A can be Brute, Krum, Median, etc
  - Bulyan is a strong Byzantine-resilience algorithm
- Bulyan – step one
  - $n \geq 4f + 3$
  - A two step process
  - The first one is to recursively use A to select $\theta = n - 2f$ gradients
    - With A, choose, among the proposed vectors, the closest one to A's output (for Krum this would be the exact output of A).
    - Remove the chosen gradient from the received set and add it to the selection set S
    - Loop back to step 1 if $|S| < \theta$.
  - $\theta = n - 2f \geq 2f + 3$, thus $S = (S1, \cdots, S\theta)$ contains a majority of non-Byzantine gradients
  - For each $i \in [1..d]$, the median of the θ coordinates i of the selected gradients is always bounded by coordinates from non-Byzantine submissions
- Bulyan – step two
  - The second step is to generate the resulting gradient F = (F[1], · · · , F[d]).
  - $\forall i \in [1..d], F[i] = \frac{1}{\beta}\sum_{X \in M[i]} X[i]$

- $\beta = \theta - 2f \geq 3$
- $M[i] = argmin_{R \subset S, |R|=\beta}(\sum_{X \in R}|X[i] - median[i]|)$
- $median[i] = argmin_{m=Y[i], Y \in S}(\sum_{Z \in S}|Z[i] - m|)$
- Each ith coordinate of F is equal to the average of the β closest ith coordinates to the median ith coordinate of the θ selected gradients.

- GuanYu
  - Byzantine tolerant learning algorithm that is
    - Resilience to Byzantine workers
    - Resilience to Byzantine parameter servers
  - GuanYu tolerates up to 1/3 Byzantine servers and 1/3 Byzantine workers
  - GuanYu uses a GAR for aggregating workers' gradients and Median for aggregating models received from servers
- Assumptions and notations
  - Asynchronous network: the lack of any bound on communication delays
  - Synchronous training: bulk-synchronous training
    - The parameter server does not need to wait for all the workers' gradients to make progress, and vice versa
    - The quorums indicate the number of messages to wait before aggregating them
  - $n_{ps} \geq 3f_{ps} + 3$ the total number of parameter servers, among which f_{ps} are Byzantine
  - $n_{wr} \geq 3f_{wr} + 3$ the total number of workers, among which f_{wr} are Byzantine
  - M the coordinate-wise median (used in both workers and servers)
  - F the GAR function (used in the servers)
  - $2f_{ps} + 3 \leq q_{ps} \leq n_{ps} - f_{ps}$ the quorum used for M
  - $2f_{wr} + 3 \leq q_{wr} \leq n_{wr} - f_{wr}$ the quorum used for F
  - d the dimension of the parameter space R^d
- GuanYu algorithm – step 1
  - At each step t, each non-Byzantine server *i* broadcasts its current parameter vector w_i^t to every worker
  - Each non-Byzantine worker *j* aggregates with M the q_{ps} first received w^t
  - And computes an estimated G_j^t of the gradient at the aggregated parameters
- GuanYu algorithm – step 2
  - Each non-Byzantine worker *j* broadcasts its computed gradient estimation G_j^t to every parameter server
  - Each non-Byzantine parameter server *i* aggregates with F the q_{wr} first received G^t
  - And performs a local parameter update with the aggregated gradient, resulting in $\overline{w}_i^t$
- GuanYu algorithm – step 3
  - Each non-Byzantine parameter server *i* broadcasts $\overline{w}_i^{t+1}$ to every other parameter servers
  - They aggregate with M the q_{ps} first received $\overline{w}_{\mathrm{k}}^{t+1}$
  - This aggregated parameter vector is $\overline{w}_i^{t+1}$