L46 FedMA Project

Halfdan Holm, Muhammad H. Sajjad, Matevz Matjasec

Abstract

In federated learning client models may diverge during training. FedMA is an approach of mitigating weight divergence by merging such models by permuting hidden representations. In this project, we investigate the performance of FedMA as an alternative to FedAvg algorithm. Further, we propose a novel algorithm Simple-FedMA that extends a simple version of FedMA to transformers. We find that SGD is a preferred optimizer as compared to Adam when using FedAvg or FedMA. Additionally, FedMA beats FedAvg in most circumstances, the difference is even more evident as the number of clients is increased. We show that Simple-FedMA can improve performance significantly, thus effectively mitigating weight divergence.

1 Introduction

Training a model using federated learning consists of two phases: 1) training the clients with their local datasets and 2) the server obtaining the locally trained model and aggregating them to obtain the final model. The most common algorithm for combining the locally trained models is FedAvg, which consists of element-wise averaging of the weights of the locally trained models [10].

A potential problem of this approach is that neural networks have several permutation symmetries in the weights. That is, for a given neural network we can find many other networks that produce the same outputs having different permutations of neuron weights. In federated learning, the locally trained networks may not choose the same set of neurons for representing the key features of the dataset. This characteristic of the neural networks causes FedAvg to often lose a significant amount of accuracy because the element-wise averaging may be performed among neurons that represent completely different features of the dataset. This phenomenon is termed weight divergence.

To counter the discussed limitations of FedAvg, a new model aggregation technique has been developed known as Federated Matched Averaging(FedMA) [15]. FedMA consists of the layer-wise matching of the neurons, i.e. given L1 and L2 as two copies of a layer in two locally trained networks, FedMA changes the order of the neurons of L2 such that it most closely matches with L1. Only after performing the matching, averaging is performed. The principle behind this approach is that the more closely matching weights are more likely to represent the same features. In fact, FedMA has shown considerably better performance as compared to FedAvg.

In this project we aim to test the performance of FedMA under new experiments that have not been performed in the original paper. For instance, the paper does not analyse FedMA on the Adam optimizer which is the most popular optimizer in MLPs and CNNs. We also want to understand how increasing the width of the network layers impacts the performance because the more the number of neurons in a layer, the more permutations we have. Finally, another objective of this project is to extend FedMA for transformers. Inspired by FedMA we develop a novel simplified algorithm Simple-FedMA. To our knowledge, this is the first implementation of a more complex weight averaging algorithm that mitigates weight divergence on a transformer neural network. Our final aim is to examine the performance of the Simple-FedMA and the nature of the weight divergence in transformer model.

2 Background

2.1 Matching on MLPs

In FedMA [15] and Git Re-Basin [1], they merge models by first permuting hidden representations to match up features. We will describe this process for MLPs.

First of all, in MLPs there are only certain nodes that can be permuted with each other, see Figure 11

$$W_1^A \Pi = W_1^B \tag{1}$$

$$\Pi W_2^A = W_2^B \tag{2}$$

The permutation of nodes in a neural network can also be represented in terms of matrix multiplication. For an MLP with one hidden layer we want to solve equation 1 for Π, as defined in [15]. Equivalent to figure 12 in appendix B.

However, the output nodes are also fixed so we could instead use equation 2 for Π. Equivalent to figure 13 in appendix B.

2.2 FedMA variables

When we run FedMA for matching networks in federated learning, there are the following relevant variables that we need to take into account [15]:

- The number of clients: this gives us exactly how many networks are being matched.
- Communication rounds: defines how many times the networks are retrained locally. In case of FedMA one training round is completed when each layer is frozen and the rest of the following layers are retrained. In the case of FedAvg, the communication rounds are equivalent to the number of iterations for which we do local retraining and averaging of the entire network.
- The size of the dataset for each client.
- Homogeneous vs Heterogeneous data distribution

2.3 Transformers

The Transformer architecture is a neural network architecture first introduced in the paper "Attention Is All You Need" by Google researchers in 2017 [14]. The architecture is based on the idea of self-attention, where the model learns to weigh the importance of different parts of the input when making predictions. This is in contrast to previous architectures such as the recurrent neural network (RNN) and the convolutional neural network (CNN), which rely on a shorter context windows to make predictions and cannot capture longer-term dependencies. The proposed architecture has been shown to achieve state-of-the-art results on a variety of natural language processing tasks, including machine translation and text summarization [12, 7]. It has also been implemented as the backbone of several popular pre-trained models such as BERT [4], GPT-3 [3] and more recently CHAT-GPT.

There are several variants of transformer architectures. In this project we rely on the PyTorch implementation from the nn.transformer module [13]. We train our network on the language modelling task with a vocabulary size of 28782 tokens. For a more detailed description of our transformer implementation see appendix C.

3 Implementation

3.1 FedMA

For this project we used the existing GitHub [9] implementation quoted by the FedMA paper. This seemed to be the most reasonable choice because it already implements the FedMA algorithm according to the specification in the paper and performs comparison with FedAvg for a given network configuration. However, using this code did not turn out to be as simple as we thought initially. The codebase produced bugs for the existing models implemented in the library which we were interested in. The bugs were essentially related to unexpected shape of the weight matrices used during layer-wise matching. Due to lack of documentation of the code, a considerable amount of time was spent on fixing such issues as the library was not very flexible about changing the architecture of an existing network. Unfortunately, due to these issues, the experiments had to be run in a limited capacity.

3.2 Simple-FedMA on transformers

To limit the scope of the project we decided to implement this only for a transformer with a single block with one head. In order to implement matching on the transformer model, we first identify where possible permutations may occur. In the end, the final permuted model needs to output the same activations as the original model for all input.

To achieve this, we answered the following questions.

- For each internal vector representation, which other vectors must share its permutation?
- For each set of nodes we want to permute, which weights should determine the permutation?
- For each vector permutation, which weights would need to be permuted to achieve that permutation?

What's important for all the permutations is that the permutation is determined based on nodes that won't be permuted later, i.e. we'd want to start permuting nodes based on weights connected to the input or output layers. Once those have been permuted, we can permute nodes connected to those nodes.

First of all we permuted the token embedding using the embedding matrix. Because of the residual stream, we had to use this same permutation throughout much of the transformer. We used it to permute the positional encoder, the input dimension of the query, key and value matrices, the embedding between the attention and the mlp, the two layer norms and the embedding before the decoder. We then split the query, key, value matrix into its three component parts. We double checked in the PyTorch source that the order of these sub-matrices was indeed q, k, v. We permuted the query and key representations based on the sum of differences in the query and key weight matrices. We permuted the value representation by using the value matrix. Finally, we permuted the hidden layer in the mlp based on the weights from the embedding to the left. All of this is visualized in Figure 1. Also, note that we used the biases to determine permutations and permuted them as appropriate. Note that all of these permutations were done by appropriately permuting the columns or rows of the matrices on the left and right of the internal vector representation.

The algorithm used to determine the permutation was a simplified version of that from [15]. We compute the cost of permuting any two nodes by the sum of the absolute difference of the weights after the permutation. This cost is stored in a cost matrix for all pairs, and the optimal permutation is then derived using a standard linear sum assignment optimiser.

We verified the correctness of our implementation by comparing the loss of the original model with the permuted model.

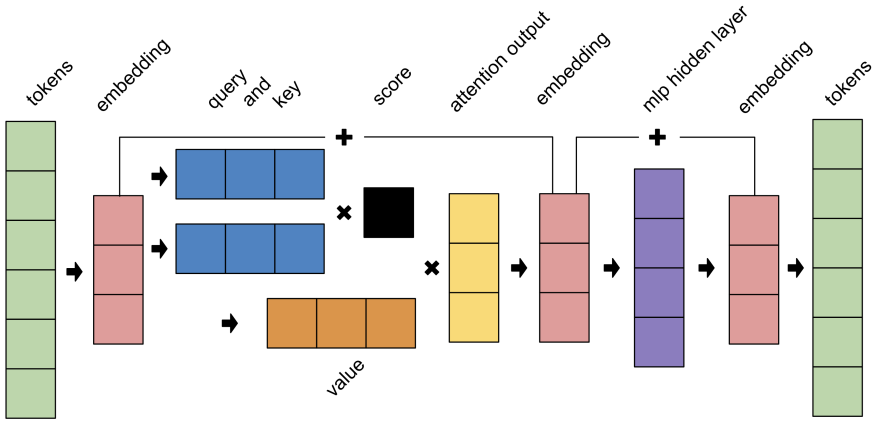

Figure 1: A visualisation of the permutations done on a single layer, single headed transformer. It visualises the vectors of the internal representations for one token. All the vectors with the same colour must be permuted in the same way. Vectors that are not in the same colour can be permuted independently. Note that the permutation happens on the weights rather than the activations. The weight matrices are represented by arrows. Some details are left out of this chart. Inspired by visualisations in [2]

We did consider how to permute attention heads as well. We would first permute the heads as a whole and then permute the inner representation of the heads, all in a similar fashion to the current permutation of the qkv matrices. However implementing this would have required significantly more work in splitting up matrices, so we leave this as future work.

In our work we only train two models in parallel and compare the loss of different merging algorithms of those models. This is a very simplified version of federated learning. We decided to stick to this model because our goal was a proof of concept for matching nodes on transformers, and it would be easier to interpret and get quick results with this framework.

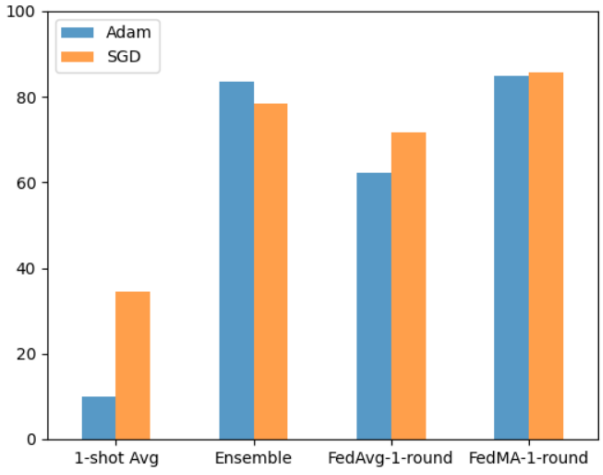

Figure 2: Test accuracy using Adam and SGD with 2 clients having heterogeneous data distribution. 1-round stands for 1 communication round between client and server.

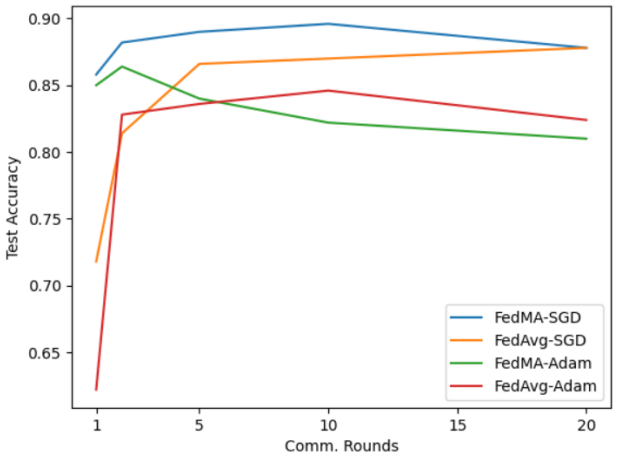

Figure 3: Test accuracy of FedMA and FedAvg under the same settings as Figure 2, only varying the number of communication rounds.

4 Experiments

4.1 FedMA on MLP

In this section, we consider some interesting experiments that have not been run in the original FedMA paper. The aim is to do a performance comparison between FedMA and FedAvg, mainly in terms of accuracy. We run these experiments on a subset of the MNIST dataset, unless otherwise specified. MNIST was chosen because almost all of the image classification experiments run by FedMA paper use Cifar-10. Unless specified otherwise, the network architecture is SimpleMLP with 4 hidden layers containing 20, 100, 120, 84 neurons respectively.

4.1.1 Adam optimizer

The FedMA paper only uses the SGD optimizer when training the network. But it is important to consider that Adam is one of the most popular optimizers and the choice of optimizer has an impact on how the weights are updated.

Figure 2 shows that one-shot averaging (simple-averaging) performs a lot worse in Adam as compared to SGD. On Adam, one-shot averaging has performance similar to a random classifier. This shows that with one-shot averaging, it is preferable to have SGD as compared to Adam. Even with one communication round, FedAvg has a considerably better performance on SGD compared to Adam. Whereas Adam outperforms SGD when we use the Ensemble of the locally trained models. The graph shows that with one communication round FedMA is marginally better on SGD as compared to Adam. Based on the graph, it can be concluded that using FedAvg, SGD is clearly a better optimizer whereas with FedMA, the choice of optimizer does not have drastic effect on the accuracy.

The analysis of figure 2 is only limited to 1 round of communication. As discussed in section 2.2, one of the variables with FedMA is the number of communication rounds. A valid question that arises here is whether at increased amount of communication FedAvg beats or matches the performance of FedMA. The FedMA paper does an analysis with varying number of communication rounds, but this analysis is limited on Cifar-10 and futhermore, it only uses SGD optimizer for this comparison.

Figure 3 shows that with a low number of communication rounds, FedMA outperforms FedAvg by a considerable margin. But as the number of communication rounds increases, FedAvg improves the performance and converges or even beats FedMA. With Adam optimizer, we can see that FedAvg clearly beats FedMA for communication rounds greater than or equal to 10. With SGD FedAvg does not beat FedMA, but as the number of communication rounds is increased, FedAvg get closer to FedMA until they converge at 20 rounds. It is surprising to see that the accuracies start to drop with both FedMA and FedAvg as the number of communication rounds is increased. A possible explanation of this is over-fitting, due to the fact that as the communication increases we are over-training the local networks on the same data. The graph also shows that SGD has a better performance with both FedAvg and FedMA.

Combining the results from figures 2 and 3, it can be concluded that in federated learning with simple

datasets such as MNIST, it is preferable to opt for SGD rather than Adam. This is due to the fact that SGD has shown to generalize better outperforming Adam on both FedMa and FedAvg. It is also observed that the smaller the number of rounds, the more evident is the difference between FedMA and FedAvg.

4.1.2 Number of clients

There is a general trend in deep learning that as more training data becomes available, the models perform better. As new clients join federated learning, the data is increased but it is still unknown what effect it has on the performance. The reason behind this is that firstly, each client has their own data distribution. Secondly, each client training the model may produce a different permutation of weights which means that combining these clients into a single model is likely to cause a higher drop in accuracy.

The experiments shown in figure 4 were run on different numbers of clients with a total of 10,000 images from the MNIST dataset. The data distribution was heterogenous and the number of communication was set to 4. It is important to note that the communication rounds do not reflect the actual number of communications of the server with one client. For instance, with FedMA using one round of communication leads to 4 communications with one client as we have 4 layers in our network. Whereas with FedAvg one round of communication leads to only one communication with the client. Hence for a fair comparison between FedMA and FedAvg, the latter should have 4 times as many rounds as FedMA. This was not an issue for the previous experiments as we were mainly comparing FedAvg with FedAvg and FedMA with FedMA for SGD and Adam. Hence in this experiment we use number of communications to refer to the amount of communication between client and server. For the experiments figure 4 this is obtained by 1 communication round for FedMA and 4 communication rounds for FedAvg. For experiments in figure 5, 20 communications are obtained by 5 communication rounds with FedMA and 20 communication rounds with FedAvg.

Figure 4 clearly shows that as the number of clients increases, the accuracy starts dropping significantly. In fact, with 10 clients, FedAvg performs similar to a random classifier and FedAvg performs just above 20%. The drop in accuracy is understandable, as training 2 networks on 10,000 datapoints would naturally lead to higher accuracy as compared to training 10 networks on the same data. So, to really analyse the performance we should increase the amount of communication between the server and the client. As a consequence, the experiment shown in figure 5 was run at 20 communications between the client and the server. This was done by setting the number of rounds to 5 and 20 respectively for FedMA and FedAvg.

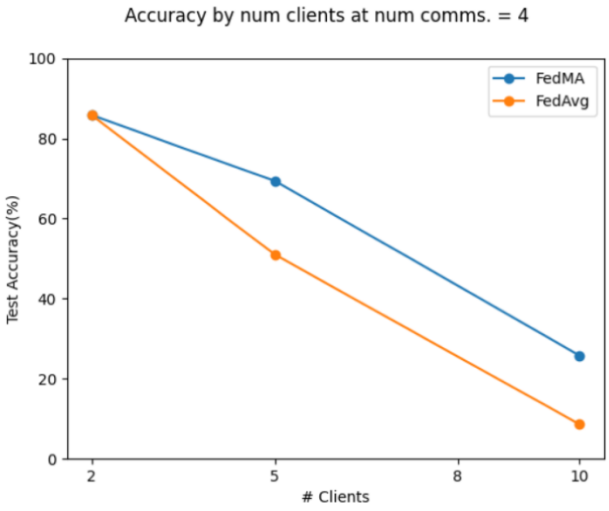

Figure 4: Test accuracy of FedMA and FedAvg on varying number of clients with number of communications with one client set to 4.

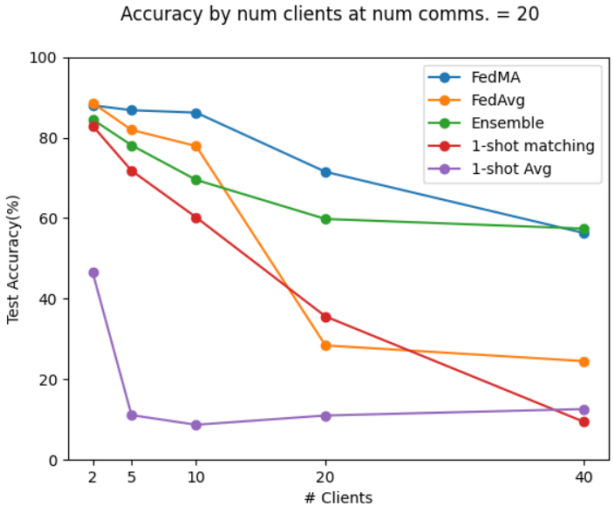

Figure 5: Test accuracy of FedMA, FedAvg, Ensemble, 1-shot matching and 1-shot averaging on varying number of clients with number of communications with one client set to 20.

Figure 5, shows a much better comparison of the accuracy between FedAvg and FedMA as the number of clients is increased. At 2 clients FedAvg beats FedMA but FedMA overtakes FedAvg afterward and as the number of clients increases, the difference between FedMA and FedAvg continues to increase. This shows that when we have many clients, the networks have very different weight distributions which makes FedAvg perform worse. Even with 20 rounds of communication, FedAvg cannot force the clients to update the same neurons for the same set of features. In fact, FedMA continues to remain the best among all methods, even beating the Ensemble.

4.1.3 Width of the layers

An important factor to consider when comparing the techniques is the width of the network i.e. the number of neurons in a given layer. As the width of the network increases, we have more and more permutations which should make it more challenging to combine the locally trained networks. In the FedMA paper, there is no analysis of how the width of the network impacts FedMA's performance. In this section, we have a performance comparison with increasing width on 5 clients with heterogeneous data distribution. As the width is increased, the training on the dataset becomes slower and slower, therefore for practical purposes the size of the dataset was restricted to 6000. The number of communication rounds was set to 1 for FedMA and 4 for FedAvg to have an equal amount of communication. The size of the first three layers varied between [20, 100, 120] and [270, 1400, 1400], increasing each layer by up to 13 times and the total size of the network by up to 67 times.

Figure 6 does not show a clear trend as figure 5 did for the number of clients. As the network size is increased, in most cases FedMA beats FedAvg. In a few exceptions FedAvg outperforms FedMA though in these cases the difference in the performance remains within 1% range. We would expect FedAvg to perform much worse as the size of layers is increased because this creates a higher number of possible symmetries. On the other hand, we do not see such a clear trend possibly due to the training dynamics and better generalization of SGD. From figure 6, we can conclude that in all the cases FedMA either outperforms FedAvg or performs very close to FedAvg. Although we would expect FedMA to be much better as the size of the network increases, this is not witnessed in the experiment.

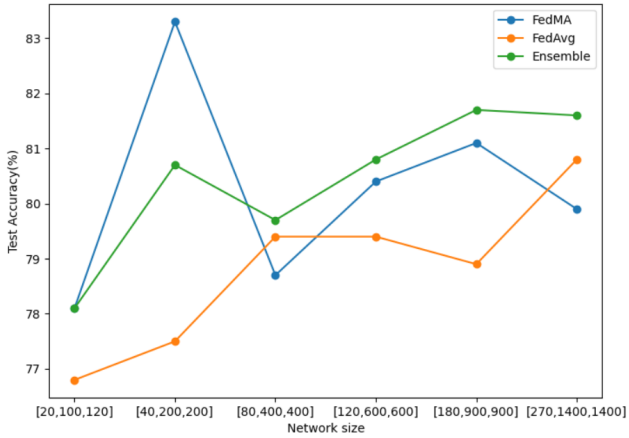

Figure 6: Test accuracy of FedMA and FedAvg on varying width of the first three hidden layers of the network.

4.2 Matching on Transformers

Experiments were done on two equally initialised Transformer networks (described in the background section) imitating two federated learning clients. The aim of the experiments was to demonstrate that the proposed Simple-FedMA algorithm on transformers can effectively mitigate weight divergence phenomenon by finding the necessary parameter permutations. Therefore the two clients were trained under constraints that ensured that the weight divergence would occur. The two networks were first trained until completion, only after which, the Simple-FedMA algorithm was applied to find the necessary permutations. The experiments investigated the effect of the choice of hyper-parameters and different initialisation on the number of permutations found by the algorithm. Two sets of models were used in the experiments. The first was the small model which has an embedding dimension of 8 and the feed-forward network twice the size of the embedding. While the second model, termed large model has the same embedding dimension, but has the hidden-layer dimension four times the size of the embedding, as proposed in the original Transformer paper. Both models had a single attention head and were composed of a single encoder block. To train the models and ensure they would exhibit weight divergence, a custom, Hetero dataset was constructed. One of the models was trained on examples from Penn TreeBank dataset [8], while the other model was trained on the same number of examples from WikiText-2 dataset [11]. The models were then evaluated on the concatenation of the two datasets. We chose to use two different datasets instead of using standard sampling techniques for generating heterogeneous datasets because we applied our transformer to a language modelling task, where there are no classes one can sample from. We did observe that at 1 epoch and learning

rate of 1 we would not get any permutations. We chose higher epoch and learning rate to get results with at least some induced permutations.

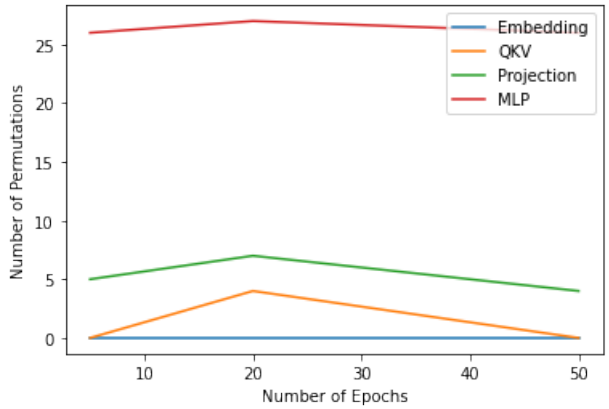

Figure 7: Breakdown of the permutations of the large model with the hidden layers 4 times the size of the embedding (as proposed in the original Transformer paper). The two networks have been trained for 5, 20 and 50 epochs respectively.

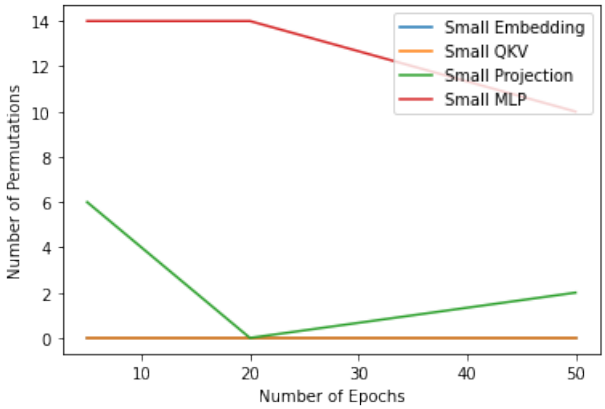

Figure 8: Breakdown of the permutations of the small model with the hidden layers two times the size of the embedding. The two networks have been trained for 5, 20 and 50 epochs respectively.

4.2.1 Number of epochs

The first investigated hyper-parameter was the number of epochs the two models were trained for. By training the two models in isolation for longer periods we expect the likelihood of weight divergence to increase due to the inherent noise of the stochastic gradient descent (SGD). Both small and large networks were trained for 5, 20 and 50 epochs respectively. Figure 7 shows the results for the large model and figure 8 shows the results for the small model. As can be seen on both figures the permutations are further broken down into four distinct groups depending on the parameter matrix at which the permutations occur. For both large and small model no permutations of the embedding matrix are observed even when models are trained in isolation for 20 epochs. Small model further has no permutations of the QKV matrix. We do however observe a significant number of permutations of the projection matrix and of the MLP layer for both models. Surprisingly, the number of permutations of the MLP layer in small models decreases as models are trained for more epochs. Figure 9 further shows the difference in loss after the permutations are applied. The smaller model constantly performs better after the permutations are applied. The positive difference in the performance of the model increases as we increase the number of epochs the two models are trained for. The large model, however, performs better after the permutations are applied only when the models are trained for 5 epochs. It should be noted that differences in performance are small.

4.2.2 Learning rate

Further, we have investigated how the choice of the learning rate hyper-parameter effects the number of found permutations. To ensure the models would exhibit weight divergence, the base learning rate hyper-parameter across other experiments was set to 15. While we would have expected that the number of permutations would scale with higher learning rate, figure 10 suggests no clear trend.

4.2.3 Different Initialisation

Finally, we have performed an experiment in which the two models were trained from distinct rather than from the identical initialisation. This was done to simulate extreme model divergence to see how much of a loss reduction could be achieved with simple-FedMA. The largest loss difference we found was from 7.41 for averaging to 7.09 for our method. We also observed some smaller differences and negative differences, but the negative differences always significantly smaller. Further, this was the only setting under which the permutation in the embedding matrix has also occurred.

4.2.4 Discussion

We were surprised to see that we got no permutations in the embedding dimension, and very few in the query and key dimensions. This suggests that efficient implementations of simple-FedMA on transformers

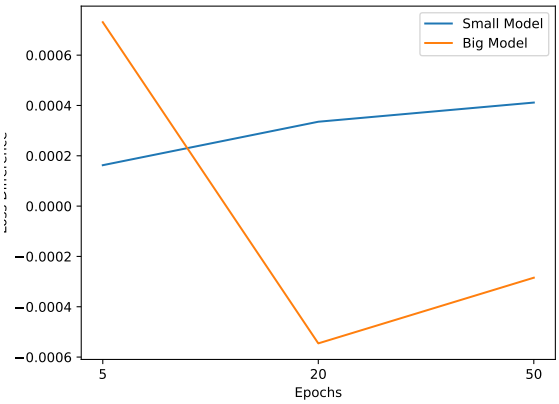

Figure 9: Comparison of the difference in loss after the applied permutations between small and large model for different number of epochs. The absolute loss for these models is around 7. The two networks have been trained for 5, 20 and 50 epochs respectively.

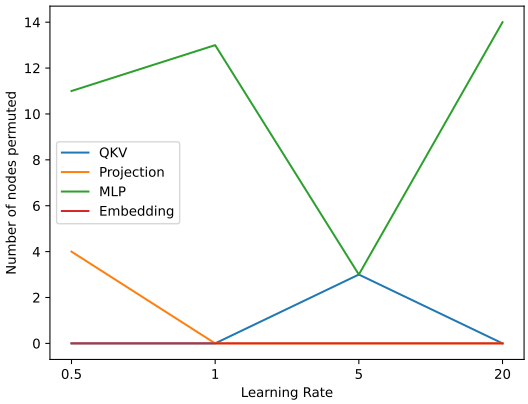

Figure 10: Effect of the learning rate on the amount of permutations. The minimum learning rate was set to 0.5, whereas the largest learning rate was set to 15.

may not need to permute these dimensions.

We also generally observed very minor performance improvements, which could suggests that these permutations are unnecessary for transformers i.e. that the details of the architecture and learning process makes it less prone to model divergence. This is supported by the fact that we were able to reduce the loss when we initialized the models differently. It could however also suggest that the freezing and retraining in FedMA is essential to improve performance on transformers. Regardless, a significant limitation of our work is that we did not apply it to a true federated learning setting with multiple communication rounds. In such a setting the improvements from the model merging can accumulate over the entire training process, which could have lead to more significant results.

It is also curious that we sometimes get a higher loss with the simple-FedMA merged model compared to the average model. This could be because we are assuming that similar weights should be connected to the same node. However, with heterogeneous datasets we could end up in situations where the same node representing the same feature simply have different weights in the two models. In that case it would make more sense to simply average the models.

5 Conclusion

We performed a comparison between FedMA and FedAvg in various settings of an MLP which have not been experimented in FedMA paper. We conclude that in federated learning, SGD optimnizer is clearly preferable to Adam because of its superior performance and better generalization with both FedMA and FedAvg. We find that with a very small number of clients (< 5) FedMA and FedAvg have a very similar performance. But as the number of clients grows, FedMA largely beats FedAvg. The difference between FedMA and FedAvg increases as the number of clients increase. Surprisingly, with increased width, we do not see such a clear trend of FedMA being better or worse than FedAvg but FedMA still beats FedAvg in most cases of increased width.

Inspired by FedMA, we have developed a novel algorithm Simple-FedMA that can be applied to neural networks with transformer architecture. Despite running the experiments in a vastly simplified federated learning setting with only 2 clients and a single communication round we have nevertheless observed increase in performance after the permutation to parameters has been applied. We hope that our research and proof of concept will inspire future work on this topic and help lead a more intelligent search through the possible design space of future weight averaging algorithms for transformers.

References

[1] Samuel K. Ainsworth, Jonathan Hayase, and Siddhartha S. Srinivasa. "Git Re-Basin: Merging Models modulo Permutation Symmetries". In: *ArXiv* abs/2209.04836 (2022).

[2] Peter Bloem. *Transformers from scratch*. 2019. URL: https://peterbloem.nl/blog/transformers.

[3] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: *ArXiv* abs/2005.14165 (2020).

[4] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *ArXiv* abs/1810.04805 (2019).

[5] Nelson Elhage et al. "A Mathematical Framework for Transformer Circuits". In: *Transformer Circuits Thread* (2021). https://transformer-circuits.pub/2021/framework/index.html.

[6] Jonathan Frankle and Michael Carbin. "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks". In: (2018). DOI: 10.48550/ARXIV.1803.03635. URL: https://arxiv.org/abs/1803.03635.

[7] Yang Liu and Mirella Lapata. "Text Summarization with Pretrained Encoders". In: *ArXiv* abs/1908.08345 (2019).

[8] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. "Building a Large Annotated Corpus of English: The Penn Treebank". In: *Comput. Linguistics* 19 (1993), pp. 313–330.

[9] Steve Martinelli and Wang Hongyi. *IBM/FedMA*. 2020. URL: https://github.com/IBM/FedMA.

[10] H. Brendan McMahan et al. "Communication-Efficient Learning of Deep Networks from Decentralized Data". In: (2016). DOI: 10.48550/ARXIV.1602.05629. URL: https://arxiv.org/abs/1602.05629.

[11] Stephen Merity et al. "Pointer Sentinel Mixture Models". In: *ArXiv* abs/1609.07843 (2016).

[12] Nathan Ng et al. "Facebook FAIR's WMT19 News Translation Task Submission". In: *ArXiv* abs/1907.06616 (2019).

[13] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[14] Ashish Vaswani et al. "Attention is All you Need". In: *ArXiv* abs/1706.03762 (2017).

[15] Hongyi Wang et al. *Federated Learning with Matched Averaging*. 2020. DOI: 10.48550/ARXIV.2002.06440. URL: https://arxiv.org/abs/2002.06440.

A Future work

An interesting area to investigate is the application of the lottery ticket hypothesis [6] with iterative magnitude pruning on the model before local training and then aggregating the model with FedAvg and FedMA. The lottery ticket hypothesis is well known for removing a considerable amount of unnecessary weights, often more than 90%, from the original network. This will cause the network to have a lot smaller amount of permutational symmetries. Furthermore, the network will not be fully connected anymore, which could lead to enforcing that certain neurons represent a set of key features in the same order across different clients. If this is the case then we expect FedAvg to improve performance on the pruned network as compared to the unpruned one. It would be interesting to compare the FedAvg with FedMA under these circumstances. Future work should also combine our work on transformers with the full FedMA implementation with freezing and extra training, support multiple attention heads and multiple decoder blocks.

B Permutations in MLPs

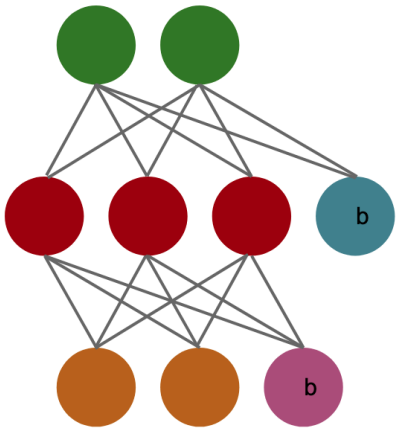

Figure 11: The colors determine the sets within which nodes can permuted. Input and output nodes should naturally not be permuted. Direction is bottom to top.

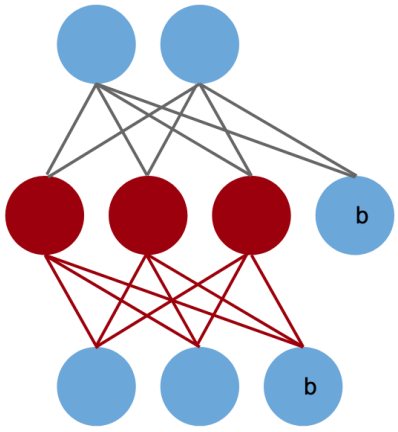

Figure 12: Test accuracy of FedMA and FedAvg under the The red connections indicate the weights that should be used to determine the permutation of the red nodes if we want to base it on the input nodes. Direction is bottom to top.

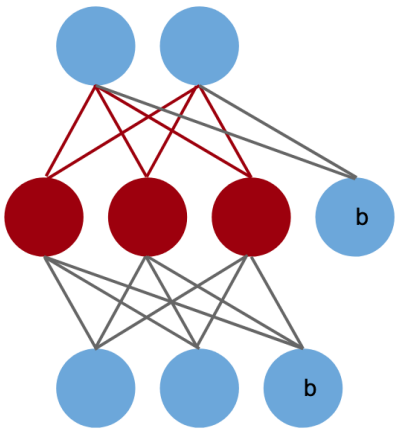

Figure 13: The red connections indicate the weights that should be used to determine the permutation of the red nodes if we want to base it on the output nodes. Direction is bottom to top.

C Our Transformer

Our transformer consists of an embedding layer followed by a positional encoding. Embedding and positional encoding matrices are the first two sets of parameters the model learns. The output is then propagated through the attention block with a single head. While the original transformer paper proposed multiple attention heads working in parallel in a single attention block, we decided to use a single attention head to simplify the matching algorithm [14]. This attention block allows the model to selectively pay attention to certain parts of the input and ignore others and this is the main contribution of [14]. This can be attributed to the third concatenated matrix of learned parameters, namely the QKV matrix. The input of the attention block is multiplied with Query, Key and Value matrix respectively obtaining Query, Key and Value vector for each token in the input. The scalar product between the Query and Key vector produces the score, which is, when further normalised, a measure of "token importance" in a given context. Finally, the score of each token in input is multiplied by the value vector of the token currently processed to obtain the output vector of the attention block. This vector is then multiplied with a fourth matrix of learned parameters termed the Projection matrix that projects the output of the attention block into an input that is most suitable for a feedforward neural network. Every layer within Transformer adds its result into a residual stream which is a shortcut connection that allows the gradient to flow directly from the input to the output of a block, bypassing any intermediate layers. The residual stream at a given layer is simply the weighted sum of outputs from all the previous layers. [5] The output of the attention block is therefore summed with

the initial token embedding before finally being fed into the multilayer perceptron (MLP). This MLP is the fifth matrix of learned parameters and is followed by a final parameter matrix termed out-projection, which projects the output of the MLP back into the dimension of the model embedding. The result is once again added into the residual stream to be summed with the previous layer output. Vector then passes through the layer normalisation and the unembedding layer after which the final logits are produced.

Many such units can be stacked upon each other, and the output of the previous one becomes the input for the following block. Our implementation however is a simplified version called the auto-encoding model which involves only one such encoder block.