MAIN PROBLEM

The main problem was the implementation of Fully Connected Neural Network.

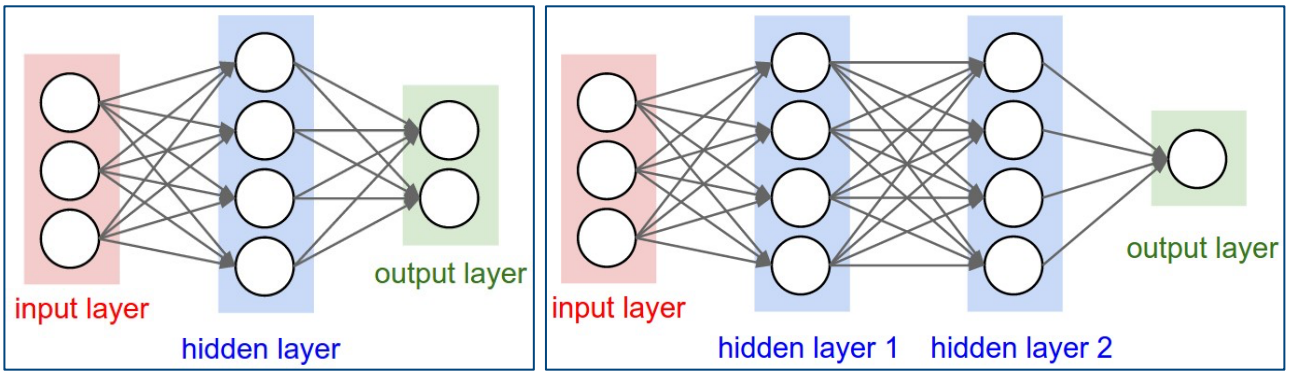


Figure 1 *Left: A 2-Layer NN; Right: A 3-Layer NN*

A fully connected neural network is a type of artificial neural network (ANN) where every neuron in one layer is connected to every neuron in the next layer. Fully connected NNs are composed of layers of neurons, with each layer connected to the next. The input layer receives the raw data, and the outputs of each subsequent layer can become inputs to other next subsequent layer. The final layer produces the output of the network.

Fully connected NNs are trained using backpropagation to minimize the error between the output of networks and the desired output. This process is repeated until the network has learned to make accurate predictions.

IMPLEMENTATION

1. Linear Layer

First, the forward pass function was implemented for a linear layer. It reshapes the input x into a vector of dimension D, multiplies it by the weight matrix w, and adds the bias vector b.

```
def forward(x, w, b):
      ...
out = x.reshape(x.shape[0], -1).mm(w) + b
...
```

As we can see from the linear layer code, the function written is a gradient function written for mini batch and tested with the input x which has shape (2,4,5,6) and contains a minibatch of 2 examples, where each example x[i] has shape (4,5,6). The result error is 3.68e-8 which is less than 1e-7 so, it is acceptable.

Then, the backward pass function was implemented. It computes the gradients with respect to x, w, and b using the chain rule.

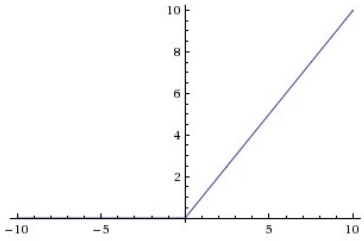
$$\frac{dL}{dx} = \frac{df}{dx} * \frac{dL}{df}$$

```
def backward(dout, cache):
    ...
    dx = dout.mm(w.t()).reshape(x.shape)
    dw = x.reshape(x.shape[0], -1).t().mm(dout)
    db = dout.sum(dim = 0)
    ...
```

The function was tested with upstream derivatives(dout) and cache(x,w,b) which was the result of forward pass. The result errors are dx error=5.22e-10, dw error=3.49e-10, db error=5.37e-10 which all are less than 1e-7 so, it is acceptable.

2. ReLU Activation

The implemented forward pass applies the ReLU activation function, which is zero when x < 0 and then linear with slope 1 when x > 0. In other words, it clamps all negative values to zero. The ReLU function gives the nonlinearity to neural network. Otherwise, there is no difference between neural network and linear matrix multiplication.



$$f(x) = \max(0, x)$$

```
    out = x.clamp(min=0)
```

After that, the backward pass for a ReLU layer was implemented. It uses the indicator function to determine which elements of the input should be propagated back.

```
    dx = (x >= 0) * dout
```

The test results are forward difference = 4.54e-9 and backward difference = 2.63e-10. They are both acceptable.

3. Sandwich Layers – Linear ReLU Activation

The Linear_ReLU layer combines the forward and backward passes of the linear and ReLU layers, it simplifies the code and makes it easier to understand and modify.

4. Loss layers: Softmax and SVM

Both Softmax and SVM were implemented in the previous homework. The softmax_loss function computes the loss and gradient for softmax classification. The svm_loss function computes the loss and gradient for multiclass support vector machine (SVM) classification.

5. Two-Layer Network

In the previous assignment, a two-layer neural network in a single monolithic class was implemented. In this assignment, the modular version of the necessary layers was implemented. The TwoLayerNet class implements a two-layer fully connected neural network with ReLU nonlinearity and SoftMax loss.

The __init__ method initializes the weights and biases of the network. The weights are initialized from a Gaussian distribution, and the biases are initialized to zero.

```
def __init__():
    ...
    self.params['W1'] = torch.randn(input_dim, hidden_dim,device=device,
dtype=dtype) * weight_scale
    self.params['b1'] = torch.zeros(hidden_dim, device=device, dtype=dtype)

    self.params['W2'] = torch.randn(hidden_dim, num_classes, device=device,
dtype=dtype) * weight_scale
    self.params['b2'] = torch.zeros(num_classes, device=device, dtype=dtype)
    ...
```

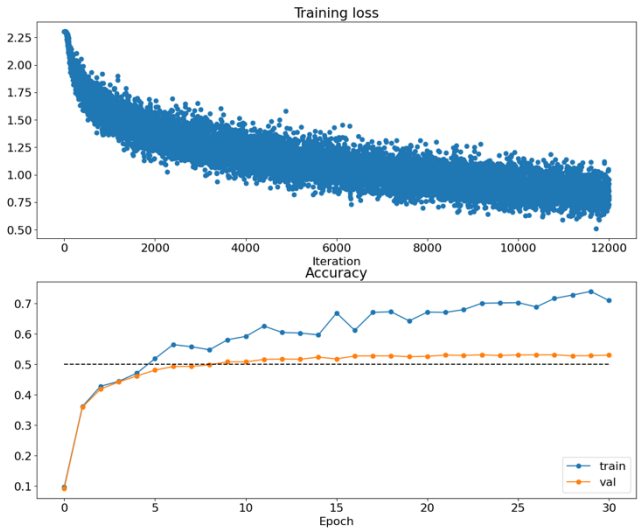
The implementation of loss function has two parts. In the first part, the outputs of previously implemented functions were computed. In the second part, these lines of code compute the loss and gradients for a two-layer neural network. The loss is computed using the SoftMax loss function, and the gradients are computed using the backward propagation algorithm.

```
def loss(self, X, y=None):
    ...
    # First Part
    l1_out, l1_cache=Linear_ReLU.forward(X,self.params['W1'], self.params['b1'])
    scores, l2_cache = Linear.forward(l1_out, self.params['W2'],
self.params['b2'])
    # Second Part
    loss, dout = softmax_loss(scores, y)
    reg = self.reg * (torch.sum(torch.square(self.params['W1'])) +
torch.sum(torch.square(self.params['W2'])))
    loss += reg

    dx_l1, dw_l1, db_l1 = Linear.backward(dout, l2_cache)
    grads['W2'] = dw_l1 + 2 * self.params['W2'] * self.reg
    grads['b2'] = db_l1

    dx_l2, dw_l2, db_l2 = Linear_ReLU.backward(dx_l1, l1_cache)
    grads['W1'] = dw_l2 + 2 * self.params['W1'] * self.reg
    grads['b1'] = db_l2
```

6. Solver

Solver was used to train a TwoLayerNet that achieves at least 50% accuracy on the validation set. The learning rate and number of epochs parameters were explored in this part of the homework. Default values of other parameters were retained. As in seen below, learning rate: 1e-1 and num_epochs = 30 was the best performed to train TwoLayerNet.

```
• learning rate: 1e-1, num_epochs = 10 ->train acc: 0.592000; val_acc: 0.508200
• learning rate: 1e-1, num_epochs = 20 ->train acc: 0.672000; val_acc: 0.526500
• learning rate: 1e-1, num_epochs = 30 ->train acc: 0.710000; val_acc: 0.530500
• learning rate: 1e-2, num_epochs = 10 ->train acc: 0.354000; val_acc: 0.343700
• learning rate: 1e-2, num_epochs = 20 ->train acc: 0.405000; val_acc: 0.381400
• learning rate: 1e-2, num_epochs = 30 ->train acc: 0.403000; val_acc: 0.407300
• learning rate: 1e-3, num_epochs = 10 ->train acc: 0.192000; val_acc: 0.179900
• learning rate: 1e-3, num_epochs = 20 ->train acc: 0.152000; val_acc: 0.160400
• learning rate: 1e-3, num_epochs = 30 ->train acc: 0.149000; val_acc: 0.161200
```



This loss function appears reasonable; in fact, based on its rate of decay, it may suggest a learning rate which appears a little bit too small. Additionally, when considering the cost is quite noisy, it suggests that the batch size may be a little too small.

Moreover, a small amount of overfitting is indicated by a difference in accuracy between training and validation lines. As you can see in the image above, after 10th epoch there is no remarkable increase in the validation accuracy.

Since the specified values were reached (minimum validation accuracy = %50), exploration on the parameters was not continued.

7. Multi-Layer Network

Multi-Layer Network consists of multiple layers of neurons, where each neuron in one layer is connected to every neuron in the next layer.

These lines of code initialize the weights and biases with an arbitrary number of hidden layers. The weights are initialized randomly and the biases are initialized to zero.

```
def __init__():
    ...
    layer_input_dim = input_dim
    for i, hd in enumerate(hidden_dims):
      self.params['W{}'.format(i+1)] = weight_scale * torch.randn(
                                                        layer_input_dim,
```

```
                                                                hd,
                                                                device=device,
                                                                dtype = self.dtype)
      self.params['b{}'.format(i+1)] = weight_scale * torch.zeros(hd,
                                                                device=device,
                                                                dtype=self.dtype)
      layer_input_dim = hd

    self.params['W{}'.format(self.num_layers)] = weight_scale * torch.randn(
                                                                layer_input_dim,
                                                                num_classes,
                                                                device=device,
                                                                dtype = self.dtype)
    self.params['b{}'.format(self.num_layers)] = weight_scale * torch.zeros(
                                                                num_classes,
                                                                device=device,
                                                                dtype = self.dtype)
    ...
```

The implementation of loss method in the FullyConnected has two parts. The first part implements the forward pass. It iterates Linear_ReLU.forward() through the network's layers. If dropout is enabled, the Dropout.forward() function is applied to the output scores.

```
def loss(self, X, y=None):
    ...
    # First Part
    scores = X
    layers = self.num_layers - 1
    total_cache = []

    # Input -> Linear ReLU -> Linear ReLU ->
    for lay in range(layers):
      W_index='W{}'.format(lay + 1)
      b_index='b{}'.format(lay + 1)
      scores, cache = Linear_ReLU.forward(scores, self.params[W_index],
self.params[b_index])
      total_cache.append(cache)

      if self.use_dropout:
        scores, cache = Dropout.forward(scores, self.dropout_param)
        total_cache.append(cache)

    # -> Linear ->
    lay+=1
    W_index='W{}'.format(lay + 1)
    b_index='b{}'.format(lay + 1)
    scores, cache = Linear.forward(scores, self.params[W_index],
self.params[b_index])
    total_cache.append(cache)
```

The second part implements the backward pass. It iterates Linear_ReLU.backward() through the network's layers. If dropout is enabled, the Dropout.backward() function is applied to the output gradient.

```
    # Second Part
    loss, dx = softmax_loss(scores, y)

    # Compute regularization and loss
    W_Sum = 0
    for lay in range(self.num_layers):
      W_Sum += torch.sum(torch.square(self.params['W%d' % (lay + 1)]))

    l2reg = self.reg * W_Sum
    loss = loss + l2reg

    # Compute gradients for output layer
    grads = {}
    W_index='W{}'.format(lay + 1)
    b_index='b{}'.format(lay + 1)
    dx,grads[W_index],grads[b_index]= Linear.backward(dx, total_cache.pop())
    grads[W_index] += 2 * self.reg * grads[W_index]

    # Compute gradients for hidden layers
    lay -= 1
    while lay >= 0:
      if self.use_dropout:
        dx = Dropout.backward(dx, total_cache.pop())
      W_index='W{}'.format(lay + 1)
      b_index='b{}'.format(lay + 1)
      dx, grads[W_index], grads[b_index] = Linear_ReLU.backward(dx,
total_cache.pop())
      grads[W_index)] += 2 * self.reg * grads[W_index]
      lay -= 1
```

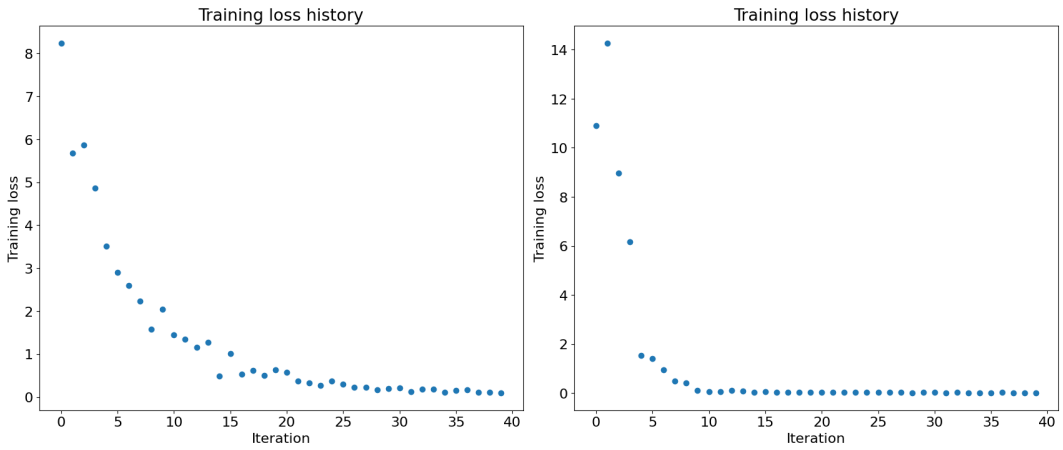


Figure 2: *Left: a three-layer network; Right: a five-layer network.*

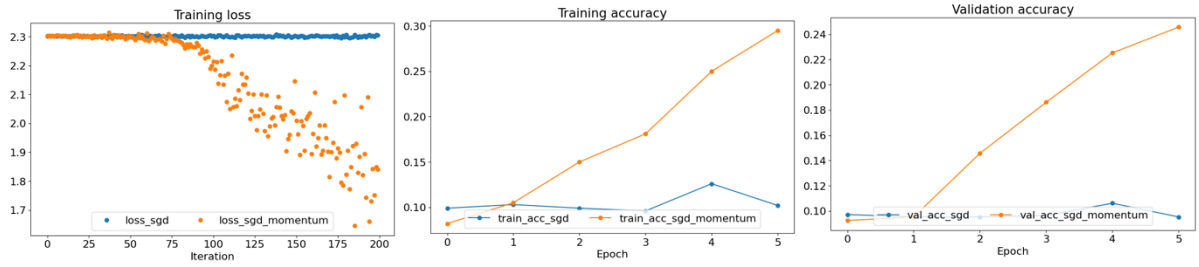
A five-layer network diverges more than a three-layer network. In other words, it decays the loss faster.

8. Update Rules

SGD + Momentum

Momentum is a technique that helps to accelerate learning by keeping track of a moving average of the gradients. The function updates the weight vector w and the velocity vector.

```
v = config['momentum'] * v - config['learning_rate'] * dw
next_w = w + v
```



SGD with momentum is more effective than vanilla SGD. This is because momentum helps to dampen oscillations in the weight updates, which can lead to faster convergence. The momentum parameter controls the strength of this dampening effect. A higher value of momentum will lead to more aggressive updates, while a lower value will lead to more conservative updates.

RMSProp

RMSProp is an optimization algorithm that improves upon the performance of SGD by adapting the learning rate for each parameter based on the history of its gradients. The algorithm maintains a moving average of the squared gradients for each parameter and uses this information to adjust the learning rate accordingly.

The below code calculates the squared gradient and updates the moving average of squared gradients and lastly calculates the updated weight vector next_w.

```
grad_squared = 0
config['cache'] = config['decay_rate'] * config['cache'] +
                                  (1 - config['decay_rate']) * torch.square(dw)
next_w = w - config['learning_rate'] * dw / (torch.sqrt(config['cache']) +
                                   config['epsilon'])
```

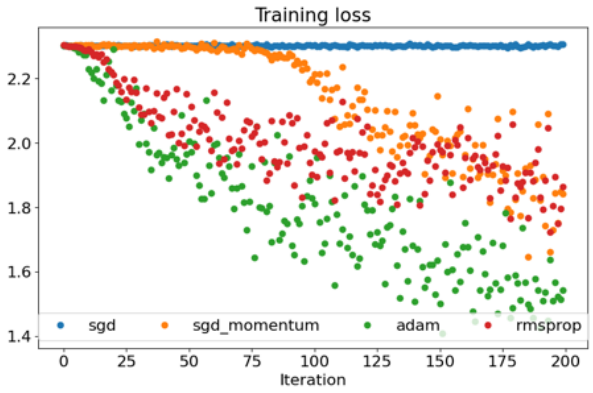
Adam

It incorporates moving averages of both the gradient and its square and a bias correction term. This allows Adam to adapt the learning rate for each parameter based on the history of its gradients and to prevent the learning rate from becoming too small or too large.

The below code increment iteration number firstly. Then it updates moving averages of gradient and squared gradient, next applies bias correction to moving averages. Lastly it calculates the updated weight vector.

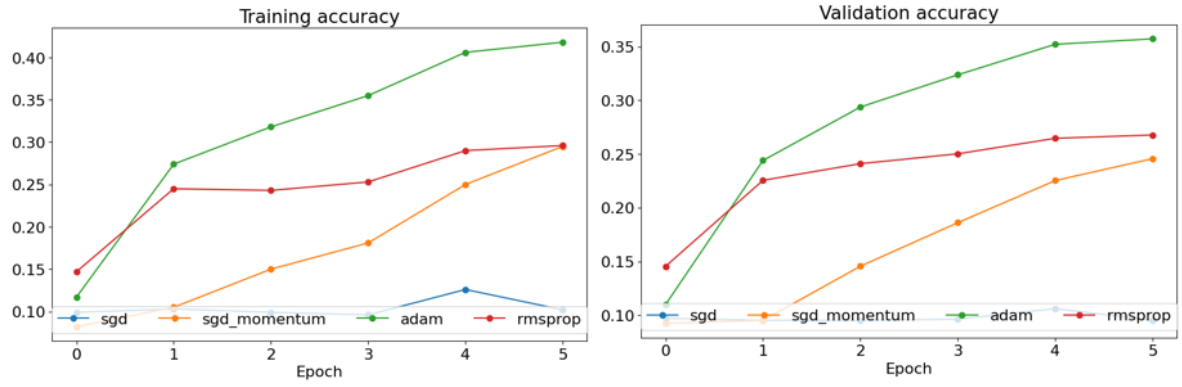
```
  config['t'] += 1
  config['m'] = config['beta1'] * config['m'] + (1 - config['beta1']) * dw
  config['v'] = config['beta2'] * config['v'] + (1 - config['beta2']) * dw * dw
  mb = config['m'] / (1 - config['beta1']**config['t'])
  vb = config['v'] / (1 - config['beta2']**config['t'])
  next_w = w - config['learning_rate'] * mb / (torch.sqrt(vb) +
config['epsilon'])
```

For Training Loss figure, we can conclude below points:



- There is noisy gradients occurrence because the training data is small.
- Learning rate is important for SGD and it seems the learning rate is too low for SGB because it doesn't even move downwards.
- Velocity term seems to work a little for SGD with momentum. But after the 100th iteration, a very oscillatory decrease is observed.
- RMSProp converges quickly at the initial steps but after that it couldn't make any further progress considering SGD with momentum.
- Adam converges to the lowest training loss most quickly.

For Training accuracy and Validation accuracy figures, we can conclude below points:



- No overfitting was observed in any method.
- We can list them according to their performance as follows:
  - Adam > RMSProp >= SGD with Momentum > SGD
- But the accuracies are not enough for CIFAR10 dataset.

9. Dropout

In the dropout section, two methods were implemented. The first method is forward pass. If the mode is 'train', then it generates a dropout mask using torch.rand(). Else the mode is 'test', then the method simply returns the input data as the output.

```
def forward(x, dropout_param):
  ...
  if mode == 'train':
    mask = ((torch.rand(x.shape) < (1 - p)) / (1 - p)).to(x.device).to(x.dtype)
    out = mask * x

  elif mode == 'test':
    out = x
```

The result of tests is shown in below:

```
Running tests with p =  0.25
Mean of input:  9.997330335850453
Mean of train-time output:  9.989851857049612
Mean of test-time output:  9.997330335850453
Fraction of train-time output set to zero:  0.2505599856376648
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.4
Mean of input:  9.997330335850453
Mean of train-time output:  9.973639588920195
Mean of test-time output:  9.997330335850453
Fraction of train-time output set to zero:  0.40133199095726013
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.7
Mean of input:  9.997330335850453
Mean of train-time output:  10.007508666610592
Mean of test-time output:  9.997330335850453
Fraction of train-time output set to zero:  0.6997119784355164
Fraction of test-time output set to zero:  0.0
```

- The mean of the train-time output decreases as p increases. This is because more neuron outputs are removed during training when the dropout probability is larger. So, the dropout layer's average output is reduced.
- The fraction of train-time output set to zero increases as p increases. This is because a higher dropout probability means that more neuron outputs are dropped during training.

The second method is backward pass. If the mode is 'train', then it multiplies the upstream derivatives by the dropout mask.

```
def backward(dout, cache):
  ...
  if mode == 'train':
    dx = dout * mask
```

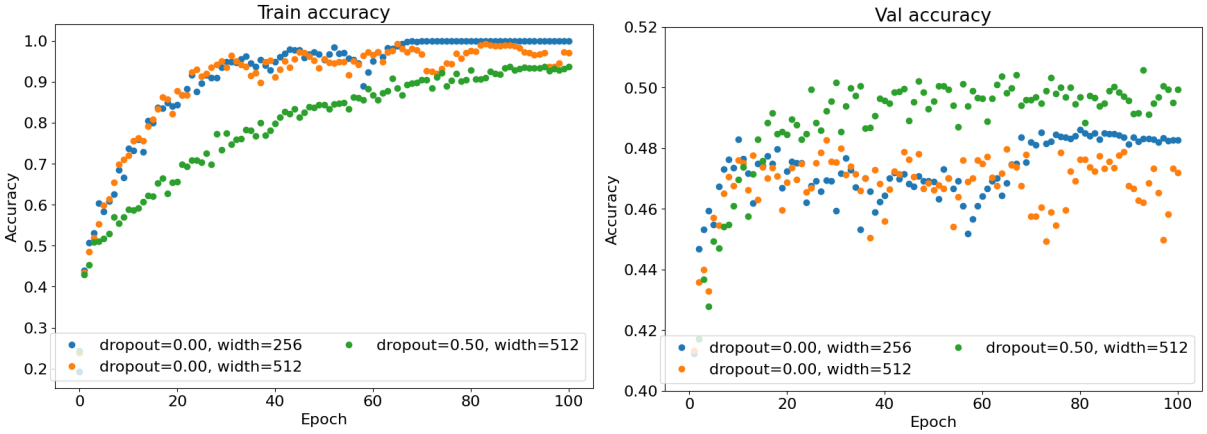
10. Fully Connected Nets with Dropout

The experiments on dropout parameter was handled and the results are below

```
Running check with dropout =  0
Initial loss:  2.3053575717037686
W1 relative error: 6.06e-08
W2 relative error: 1.02e-07
W3 relative error: 5.89e-08
b1 relative error: 1.28e-07
b2 relative error: 2.05e-08
b3 relative error: 3.41e-09

Running check with dropout =  0.25
Initial loss:  2.304579158131078
W1 relative error: 3.55e-08
W2 relative error: 5.60e-08
W3 relative error: 4.28e-08
b1 relative error: 9.91e-08
b2 relative error: 1.57e-08
b3 relative error: 3.02e-09

Running check with dropout =  0.5
Initial loss:  2.2843557967595594
W1 relative error: 9.21e-09
W2 relative error: 1.66e-08
W3 relative error: 1.40e-08
b1 relative error: 2.95e-08
b2 relative error: 1.62e-08
b3 relative error: 2.78e-09
```

- When dropout rises, the initial loss decreases. This is so that a reduced training loss can result from overfitting, which dropout helps to prevent.
- As the dropout rises, the relative errors of W1, W2, W3, b1, b2, and b3 decrease. This is because dropout contributes to the network's bias regularization, which can reduce the relative errors between the biases and the related reference values.

The result accuracy plots of regularization experiment with different hidden size and dropout are shown in below.



As we can see, the whole models have overfit problem but as the dropout increases whether hidden size changes or not the gap between training and accuracy is decreasing. In conclusion, dropout helps to prevent overfitting.

CONCLUSION

- Linear layers are a fundamental building block of neural networks. They are used to combine the outputs of previous layers and produce a new output.
- The ReLU activation is a good choice for neural networks because it is computationally efficient and produces non-negative outputs.
- Sandwich layers combines the strengths of linear layers and ReLU activations.
- Loss layers are used to measure the difference between the predicted output of a neural network and the desired output. They help to get better weights for learning.
- Two-layer networks are a simple and effective way to learn linear relationships between inputs and outputs. On the other hand, multi-layer networks can learn better complex relationships between inputs and outputs.
- The choice of update rule can have a significant impact on the performance of a neural network. Update rules are used to update the weights of a neural network during training. They are responsible for adjusting the weights in a way that minimizes the loss function. Adam can achieve high training accuracy quickly and efficiently.
- Dropout is a regularization technique that can help to prevent overfitting in neural networks.
- Fully connected nets with dropout are performed better than without dropout regularization.