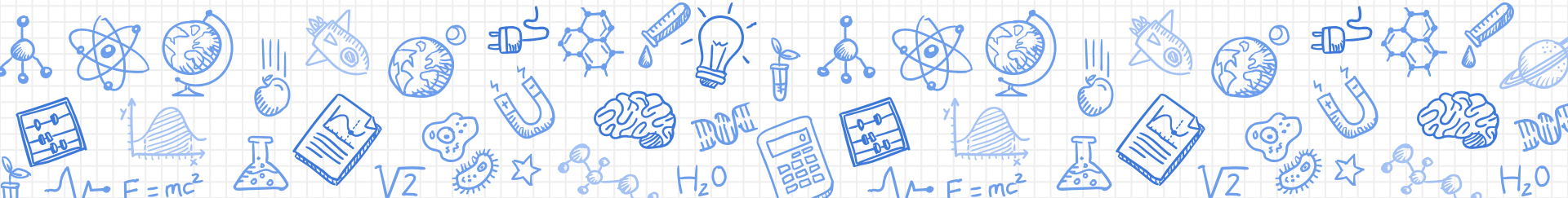
CNNs (KERAs with Tensorflow) for CIFAR

Hyperparameter and Performance of CNN

Li Sun (ls1229)

Faye Wang (yw485)

Introduction



Overview

Image classification is one of the core computer vision task that aims to assign an input image one label from a fixed set of categories.

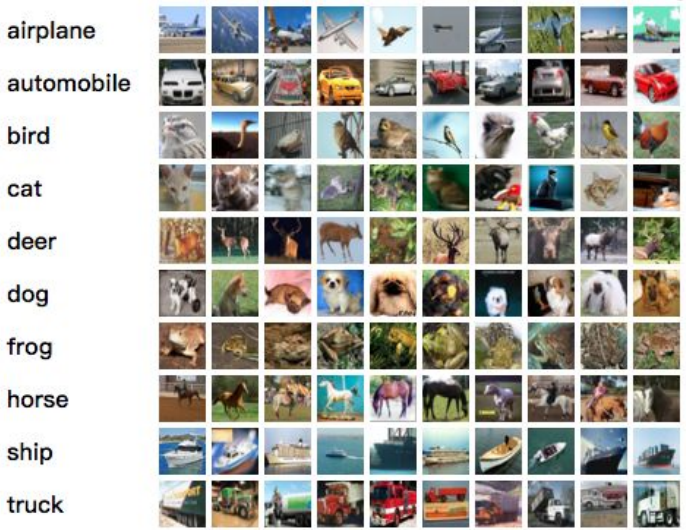
Many other seemingly distinct Computer Vision tasks (such as object detection, segmentation) can be reduced to image classification.

In the design of algorithm, it's important to choose the values of hyperparameters in order to reach the optimized result, increase training rate and reduce the test error.

Objective

- ✘ Train a network to accurately classify images.
- ✘ Deploy CNN models into real world application. (CIFAR10)
- ✘ See how hyperparameters affect training rate and test error.

Data Sets

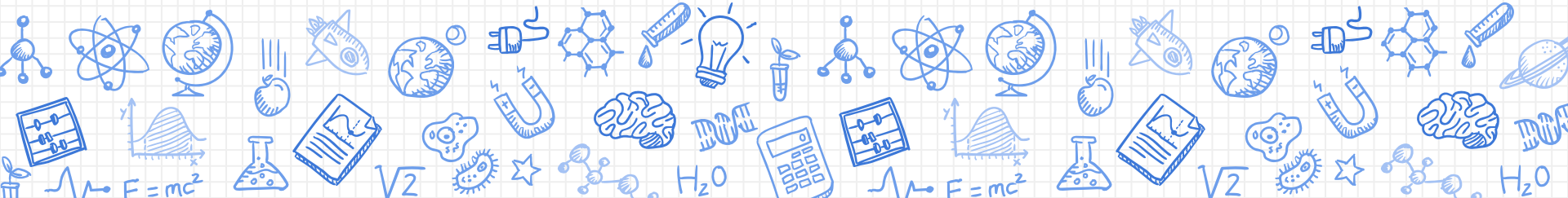


- CIFAR-10 dataset
- 60000 colour images (32x32x3)
- 10 classes

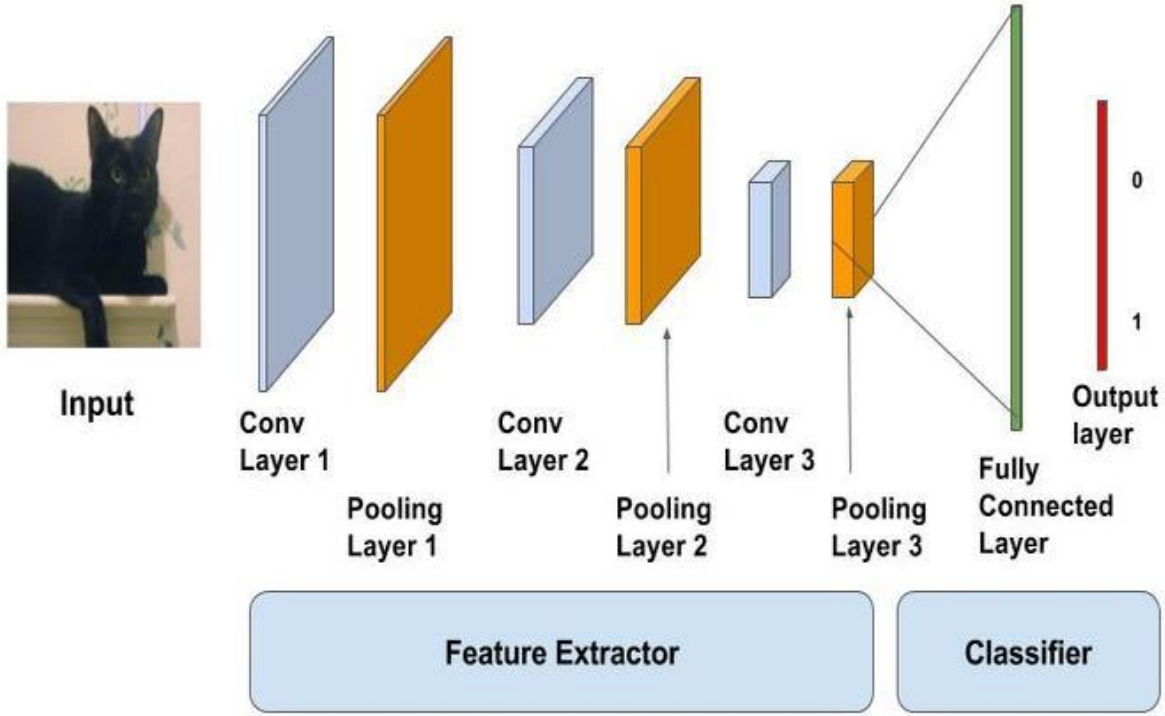
Data Sets

- ✘ 6000 images per class.
- ✘ 50000 training images and 10000 test images.

CNN Structure and Training



Convolution Neural Network



Training Neural Networks

Setup

- ✘ Layers (Conv, Max Pooling..)
- ✘ Filter size
- ✘ Activation functions
- ✘ Preprocessing
- ✘ Weight initialization
- ✘ Regularization
- ✘ Gradient checking

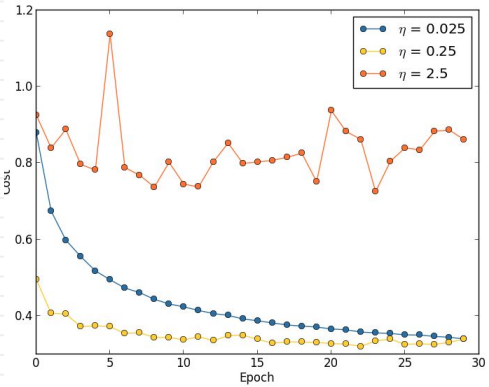
Training dynamics

- ✘ Parameter updates
- ✘ Hyperparameter optimization

Evaluation

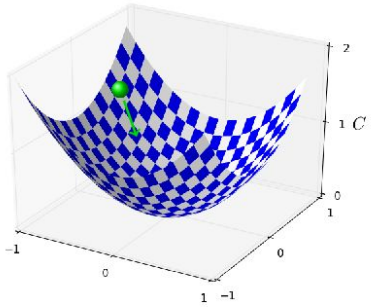
- ✘ Model ensembles
- ✘ Test error

Learning Rate



Learning rate is too high:
The parameters are bouncing around chaotically, unable to settle in a nice spot

Learning rate too low:
The speed of decay can be very slow



Optimizer

Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks.

Gradient descent optimization algorithms:

- ✘ SGD - Stochastic gradient descent
- ✘ Momentum
- ✘ Nesterov accelerated gradient
- ✘ Adam
- ✘ Adagrad
- ✘ RMSprop

Optimizer – SGD

SGD performs a parameter update for each training example at a time.

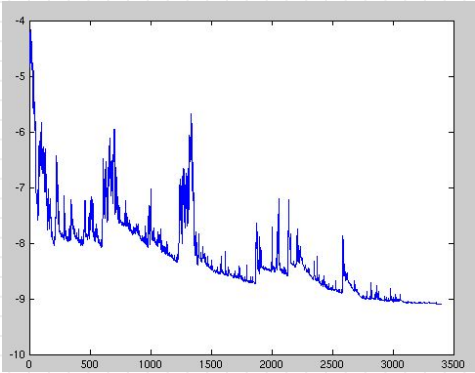
Parameter updates: $\theta = \theta - \eta \cdot \nabla J(\theta; x(i); y(i))$

η - learning rate

∇J(θ) - Gradient of Loss function-**J(θ)** w.r.t parameters-'θ'.

{x(i) ,y(i)} are the training examples

SGD is much faster than Batch gradient descent and can be used to learn online. SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily



SGD fluctuation (Source: Wikipedia)

Optimizer – Momentum

Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations $V(t) = \gamma V(t-1) + \eta \nabla J(\theta)$ $\theta = \theta - V(t)$

It adds a fraction **γ** of the update vector of the past time step to the current vector. The momentum term **γ** is usually set to 0.9 or a similar value.

The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions.
As a result, we gain faster convergence and reduced oscillation.

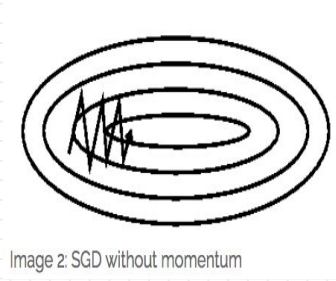


Image 2: SGD without momentum

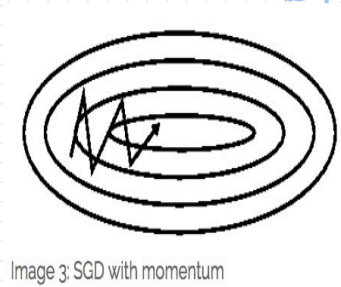


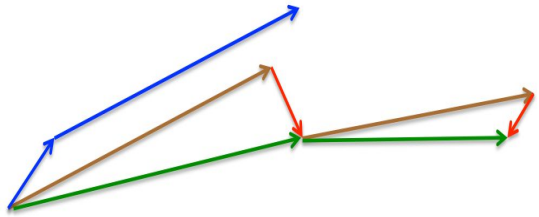
Image 3: SGD with momentum

Optimizer – Nesterov accelerated gradient

$$V(t) = \gamma V(t-1) + \eta \nabla J(\theta - \gamma V(t-1)) \qquad \theta = \theta - V(t)$$

Nesterov accelerated gradient (NAG) is a way to give our momentum term the prescience that it enables us to effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters.

Now that we are able to adapt our updates to the slope of our error function and speed up SGD in turn, we would also like to adapt our updates to each individual parameter to perform larger or smaller updates depending on their importance.



Optimizer – Adam

Adam stands for Adaptive Moment Estimation.

It computes adaptive learning rates for each parameter θi at every time step t.

Adam keeps an exponentially decaying average of past gradients M(t)

M(t) and V(t) are values of the first moment which is the Mean and the second moment which is the uncentered variance of the gradients respectively.

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}.$$

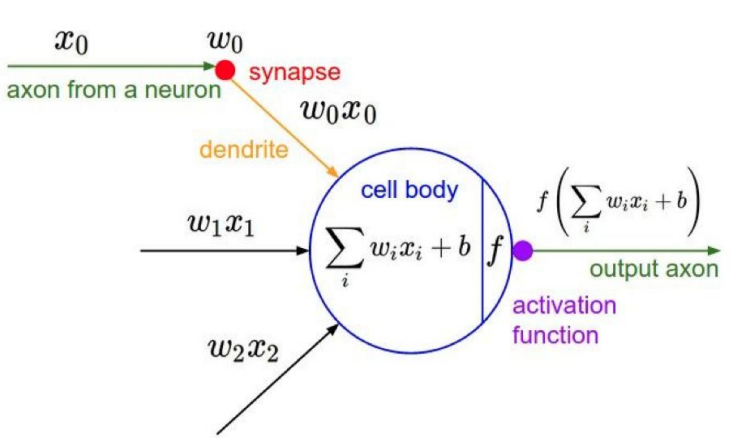
$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}.$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t.$$

The values for β1 is 0.9 , 0.999 for β2, and (10 x exp(-8)) for ϵ.

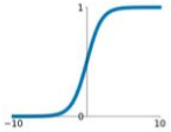
Adam works well in practice and compares favorably to other adaptive learning-method algorithms as it converges very fast and the learning speed of the Model is quiet Fast and efficient

Activation Functions



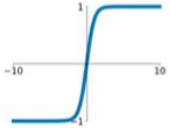
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



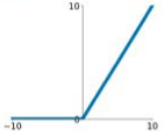
tanh

$\tanh(x)$



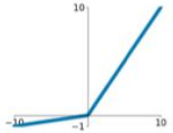
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

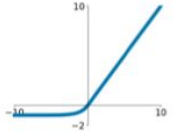


Maxout

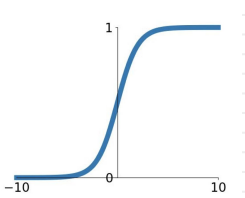
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



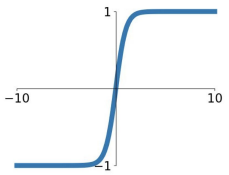
Activation Functions



Sigmoid

$$\sigma(x) = 1/(1+e^{-x})$$

- ✘ Squashes numbers to range [0,1]
- ✘ Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron
- ✘ Saturated neurons "kill" the gradients
- ✘ Sigmoid outputs are not zero-centered
- ✘ exp() is a bit compute expensive

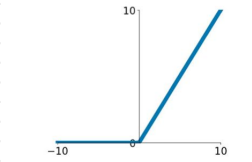


tanh(x)

$$tanh(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$$

- ✘ Faster than sigmoid
- ✘ Squashes numbers to range [-1,1]
- ✘ zero centered (nice)
- ✘ still kills gradients when saturated

Activation Functions

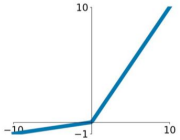


ReLU
(Rectified Linear Unit)

$$y = \begin{cases} 0 & (x \leq 0) \\ x & (x > 0) \end{cases}$$

- ✘ Does not saturate (in +region)
- ✘ Very computationally efficient
- ✘ Converges much faster than sigmoid/tanh in practice
- ✘ Actually more biologically plausible than sigmoid

- ✘ When x <0, the gradient is 0,
- ✘ Then the ReLU is dead and will not be activated.

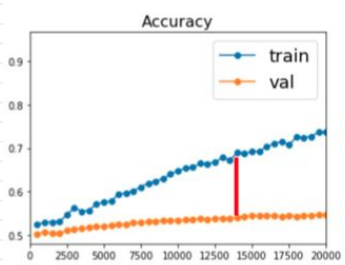


Leaky ReLU
$f(x) = \max(0.01x, x)$

- ✘ Does not saturate
- ✘ Computationally efficient
- ✘ Converges much faster than sigmoid/tanh in practice!
- ✘ will not “die”.

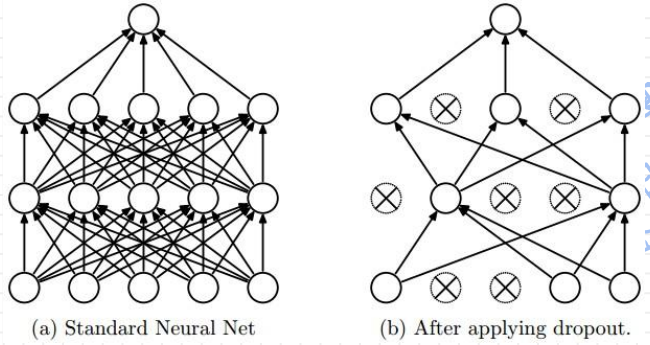
Regularization

Why? Helps reduce test error,
Prevent overfitting

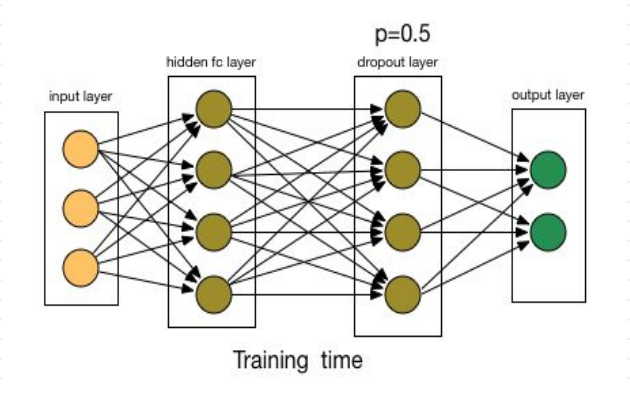


How?

- ✘ L1 Regularization $R(W) = \sum_k \sum_l W_{k,l}^2$
- ✘ L2 Regularization $R(W) = \sum_k \sum_l |W_{k,l}|$
- ✘ **Dropout**
  - ✗ extremely effective, simple
  - ✗ In each forward pass, randomly set some neurons to zero
  - ✗ Probability of dropping is a hyperparameter; 0.5 is common


(a) Standard Neural Net (b) After applying dropout.

Regularization: Dropout

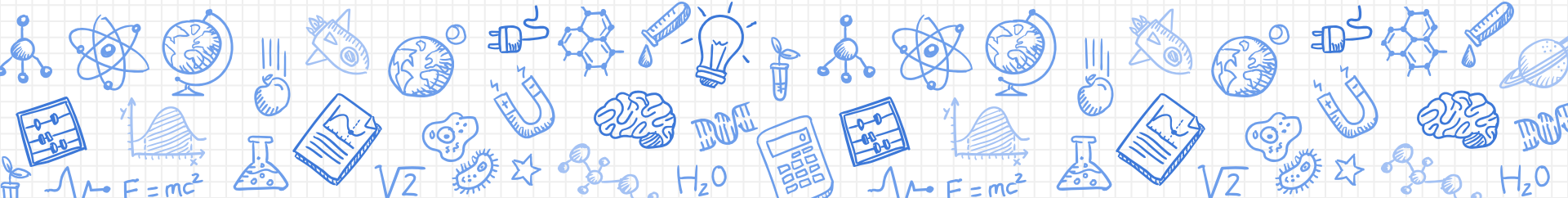


At each training iteration a dropout layer randomly removes some nodes in the network along with all of their incoming and outgoing connections.
Dropout can be applied to hidden or input layer.

Why it works?

- ✘ Nodes become more insensitive to the weights of the other nodes (co-adaptive), the model is more robust.
- ✘ Can be viewed as a form of averaging multiple models ("ensemble")
- ✘ Training a neural network is like training a collection of 2^n thinned networks with parameters sharing, where each thinned network gets trained very rarely,

Implementation



Configuration

GPU: GTX970: ~17s per epoch

CPU: i7 3770: ~390s per epoch

Around 22x faster

```
Train on 50000 samples, validate on 10000 samples
Epoch 1/100
50000/50000 [==============================] - 19s 378us/step - loss: 2.3104 - acc: 0.1001 - val_loss: 2.3089 - val_acc: 0.1000
Epoch 2/100
50000/50000 [==============================] - 17s 336us/step - loss: 2.3096 - acc: 0.0998 - val_loss: 2.3049 - val_acc: 0.1000
Epoch 3/100
50000/50000 [==============================] - 17s 336us/step - loss: 2.3088 - acc: 0.1017 - val_loss: 2.3100 - val_acc: 0.1000
Epoch 4/100
50000/50000 [==============================] - 17s 340us/step - loss: 2.3094 - acc: 0.1030 - val_loss: 2.3063 - val_acc: 0.1000
Epoch 5/100
50000/50000 [==============================] - 17s 334us/step - loss: 2.3091 - acc: 0.1016 - val_loss: 2.3083 - val_acc: 0.1000
Epoch 6/100
50000/50000 [==============================] - 16s 329us/step - loss: 2.3096 - acc: 0.0976 - val_loss: 2.3064 - val_acc: 0.1000
Epoch 7/100
50000/50000 [==============================] - 17s 335us/step - loss: 2.3091 - acc: 0.1017 - val_loss: 2.3080 - val_acc: 0.1000
Epoch 8/100
50000/50000 [==============================] - 17s 335us/step - loss: 2.3088 - acc: 0.0997 - val_loss: 2.3131 - val_acc: 0.1000
Epoch 9/100
50000/50000 [==============================] - 17s 337us/step - loss: 2.3086 - acc: 0.1014 - val_loss: 2.3169 - val_acc: 0.1000
Epoch 10/100
```

Base Model

OPERATION		DATA DIMENSIONS			WEIGHTS(N)	WEIGHTS(%)
Input	#####	3	32	32		
Conv2D	|/				896	0.1%
relu	#####	32	32	32		
Conv2D	|/				9248	0.7%
relu	#####	32	30	30		
MaxPooling2D	Y max				0	0.0%
	#####	32	15	15		
Dropout	| ||				0	0.0%
	#####	32	15	15		
Conv2D	|/				18496	1.5%
relu	#####	64	15	15		
Conv2D	|/				36928	3.0%
relu	#####	64	13	13		
MaxPooling2D	Y max				0	0.0%
	#####	64	6	6		
Dropout	| ||				0	0.0%
	#####	64	6	6		
Flatten	|||||				0	0.0%
	#####	2304				
Dense	XXXXX				1180160	94.3%
relu	#####	512				
Dropout	| ||				0	0.0%
	#####	512				
Dense	XXXXX				5130	0.4%
softmax	#####	10				

4 hidden layer:

3 conv + 1 flat

Activation: RELU

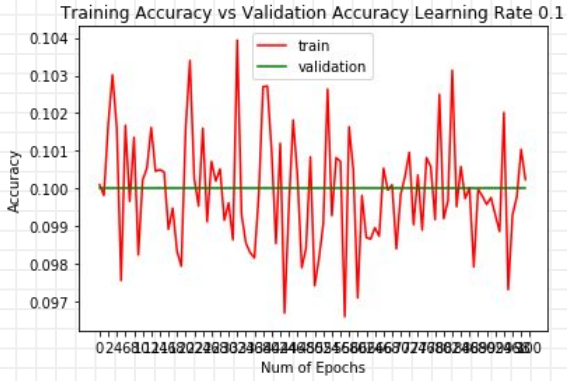
Optimizer: SGD+nesterov (1st momentum)

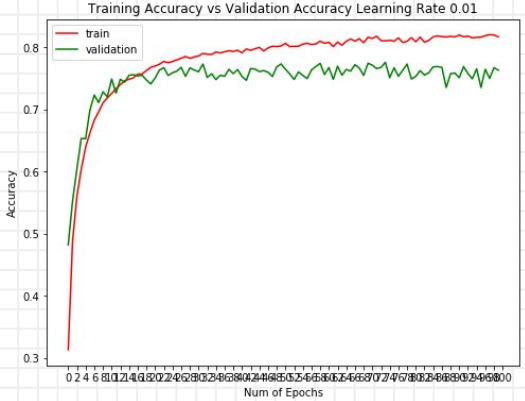
batch_size = 32

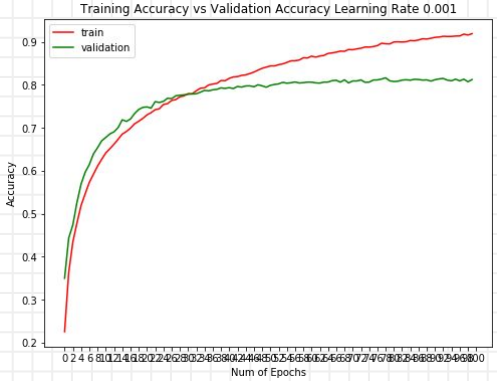
epochs = 100

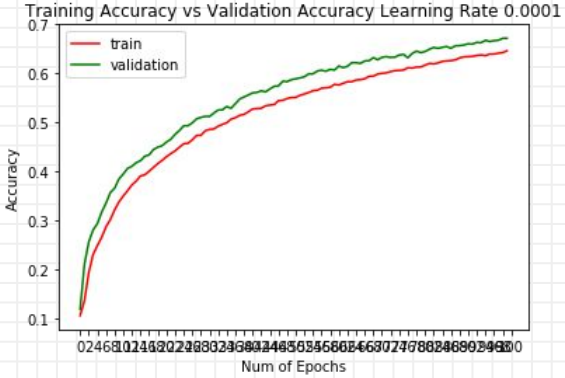
Regularization: Dp 25%

Learning Rate under SGD+nesterov momentum

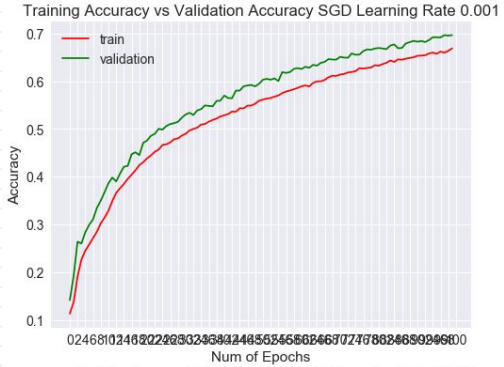


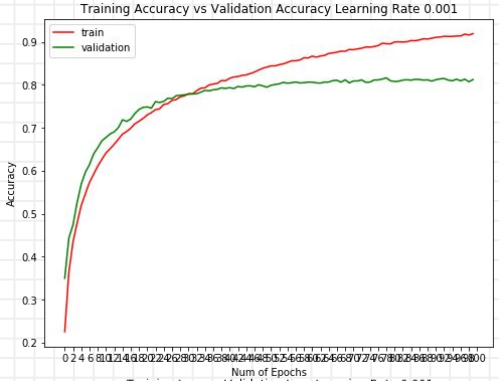


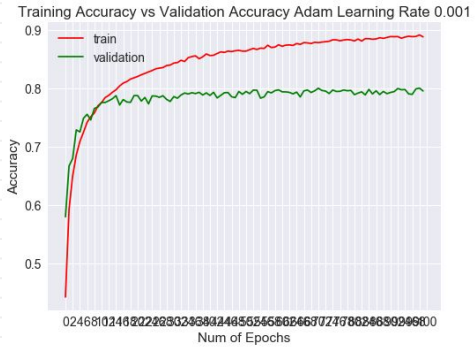


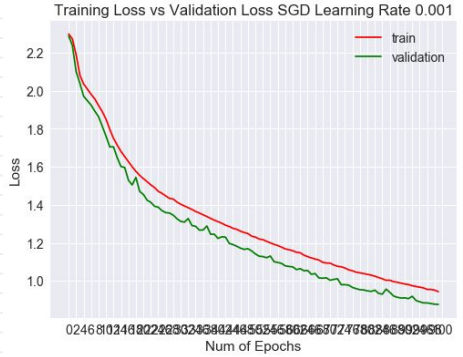


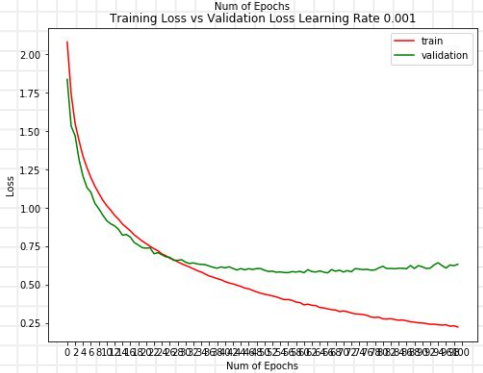
Optimizer: SGD/nesterov (1st momentum)/Aadm (1st + 2nd momentum)

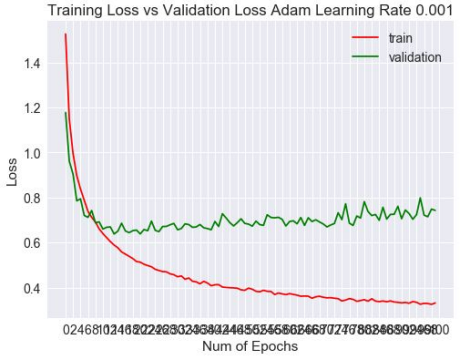










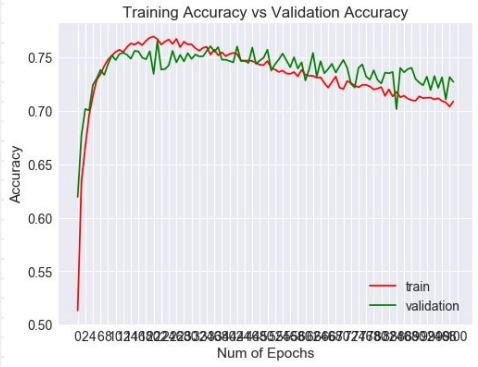


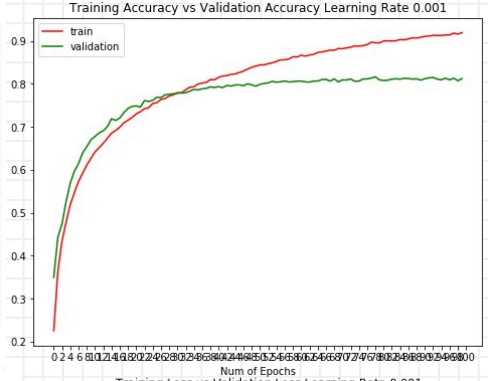
SGD

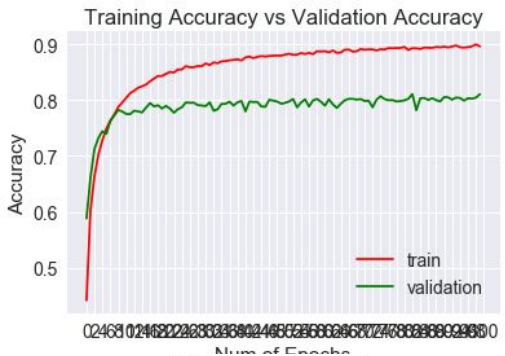
Nesterov

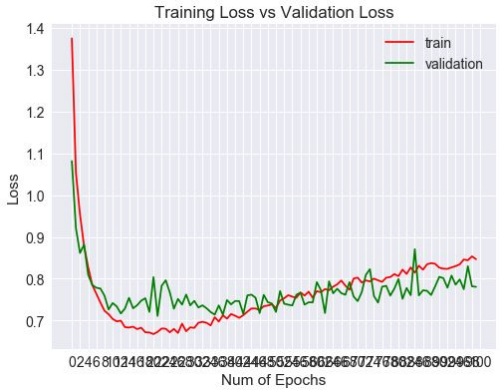
Adam

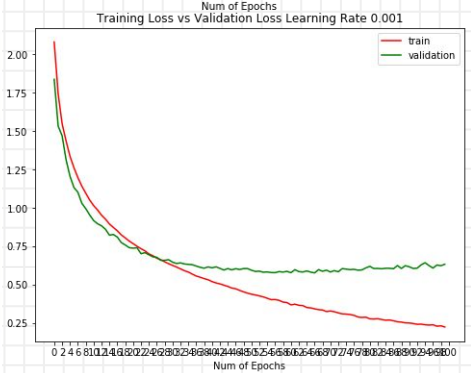
Activation Functions: PReLU/tanh/ReLU

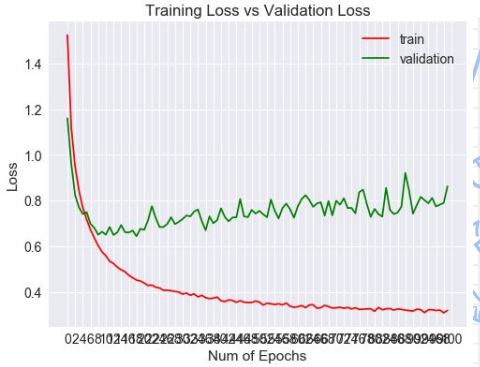












tanh ReLU PReLU

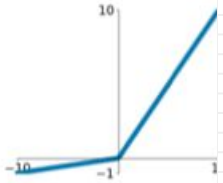
PReLU

OPERATION		DATA DIMENSIONS	WEIGHTS(N)	WEIGHTS(%)
Input	#####	3 32 32		
Conv2D	|/ ----------		896	0.1%
	#####	32 32 32		
PReLU	????? ----------		32768	2.4%
	#####	32 32 32		
Conv2D	|/ ----------		9248	0.7%
	#####	32 30 30		
PReLU	????? ----------		28800	2.2%
	#####	32 30 30		
MaxPooling2D	Y max ----------		0	0.0%
	#####	32 15 15		
Dropout	| || ----------		0	0.0%
	#####	32 15 15		
Conv2D	|/ ----------		18496	1.4%
	#####	64 15 15		
PReLU	????? ----------		14400	1.1%
	#####	64 15 15		
Conv2D	|/ ----------		36928	2.8%
	#####	64 13 13		
PReLU	????? ----------		10816	0.8%
	#####	64 13 13		
MaxPooling2D	Y max ----------		0	0.0%
	#####	64 6 6		
Dropout	| || ----------		0	0.0%
	#####	64 6 6		
Flatten	||||| ----------		0	0.0%
	#####	2304		
Dense	XXXXX ----------		1180160	88.2%
	#####	512		
PReLU	????? ----------		512	0.0%
	#####	512		
Dropout	| || ----------		0	0.0%
	#####	512		
Dense	XXXXX ----------		5130	0.4%
softmax	#####	10		

Leaky ReLU with trainable alpha:

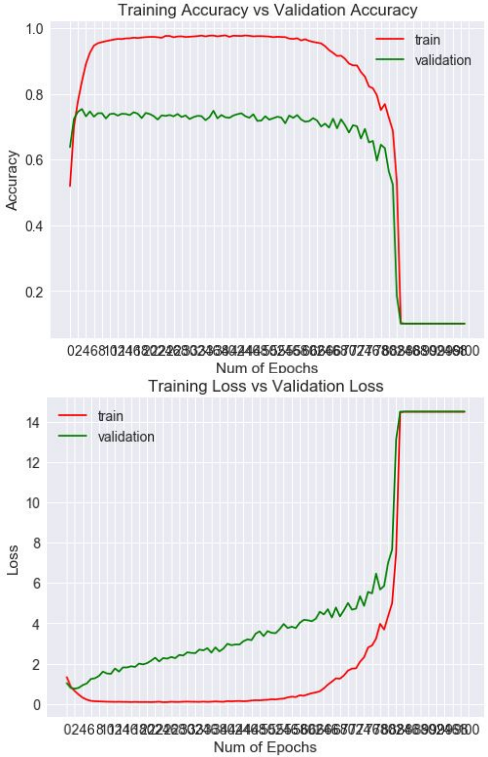
Leaky ReLU

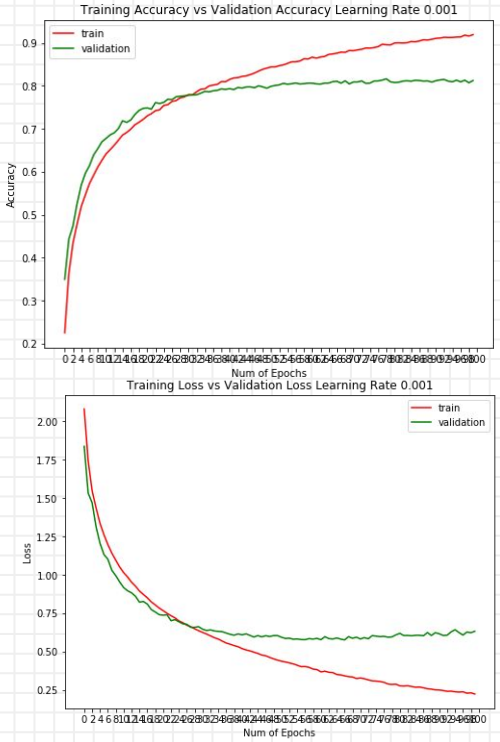
$$\max(0.1x, x)$$

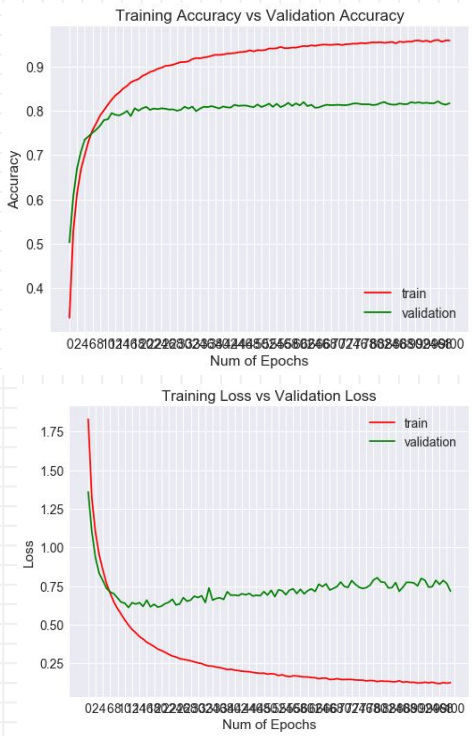


Slower than other activation functions: 7s

Regularization





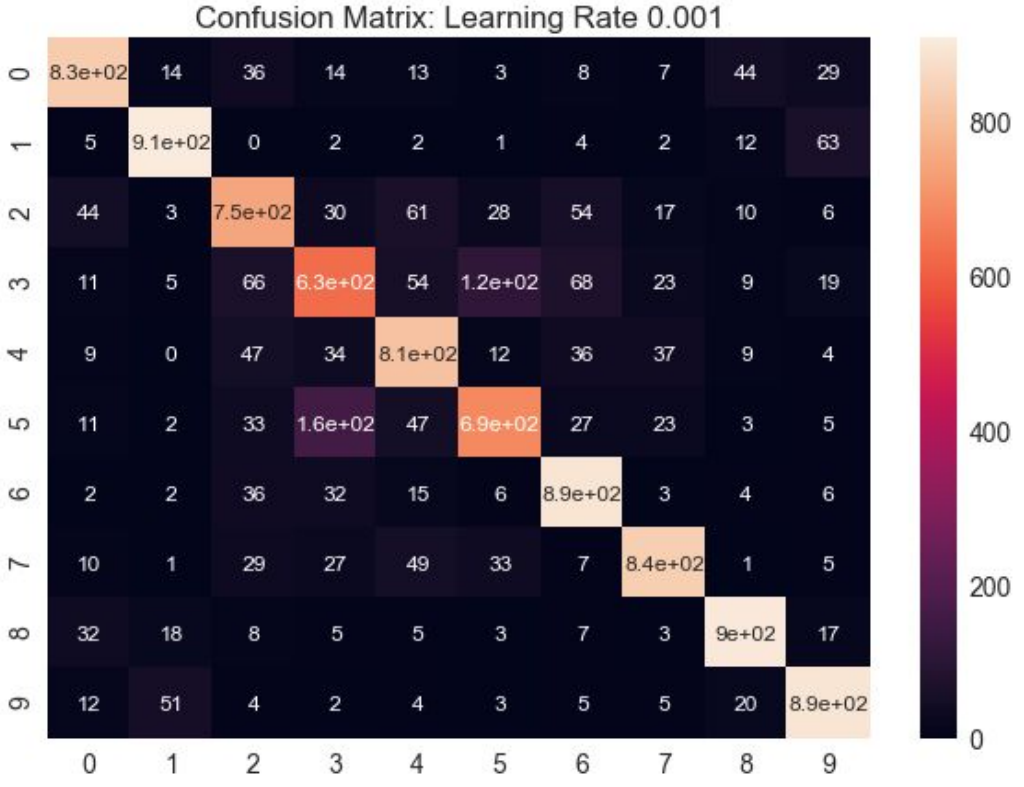


No

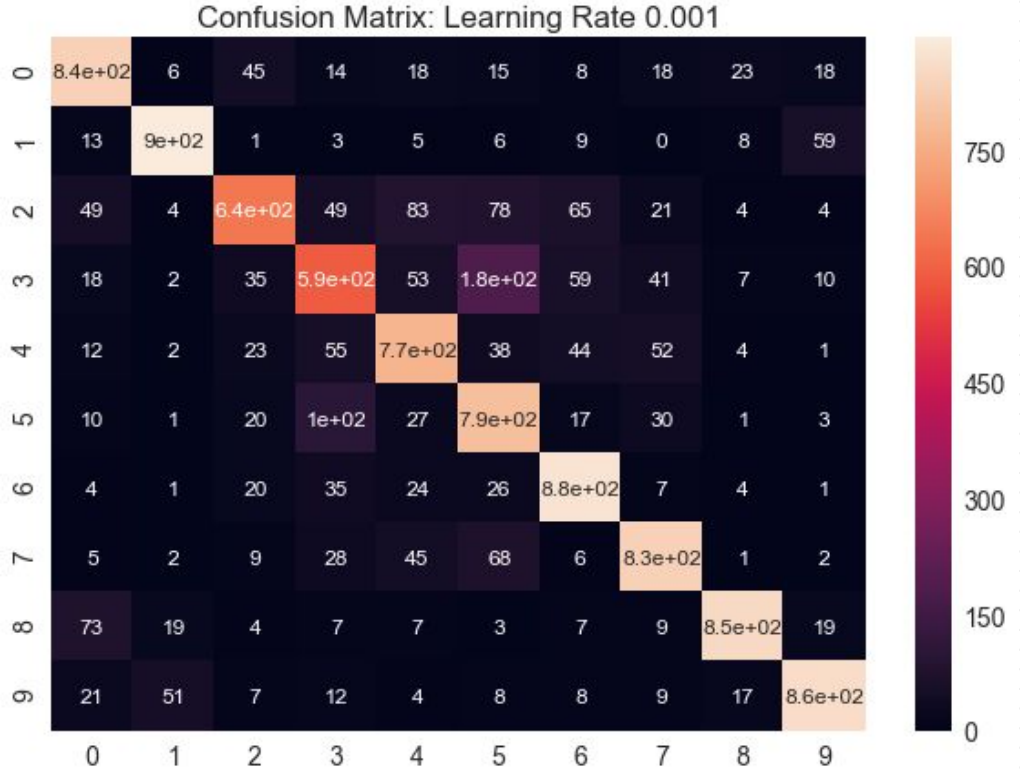
Dropout 25%

Batch Normalization+dropout

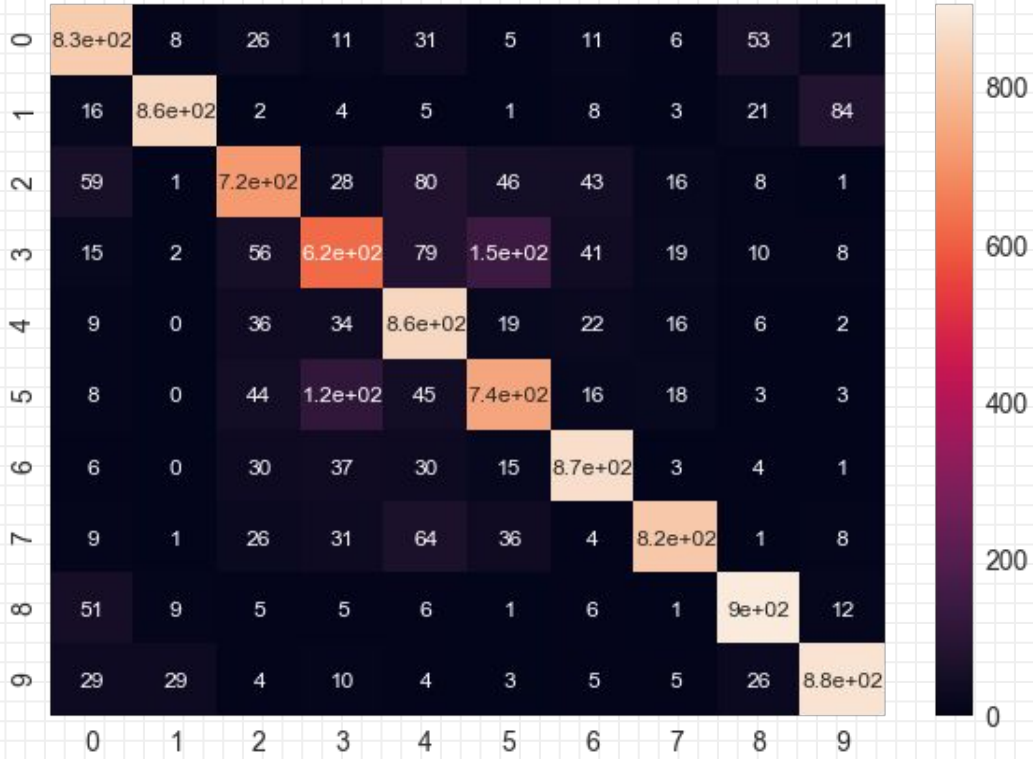
Confusion matrix: 0.001



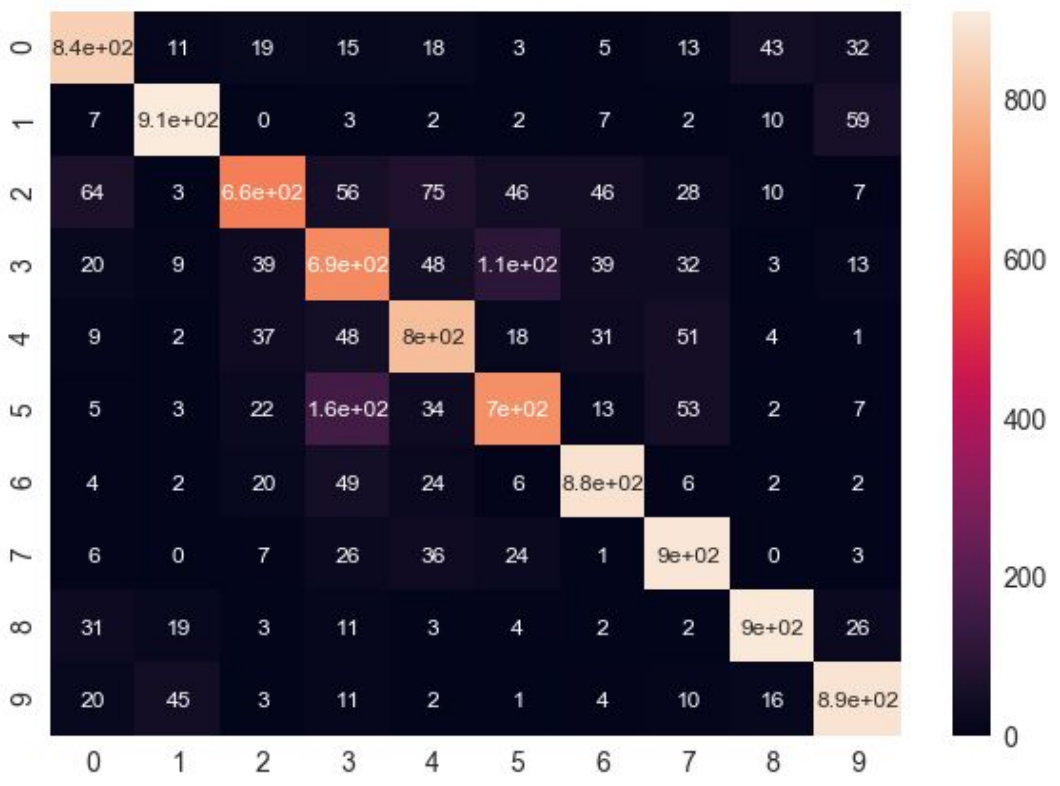
Confusion matrix: Adam+0.001



Confusion matrix: Adam+0.001+PReLU

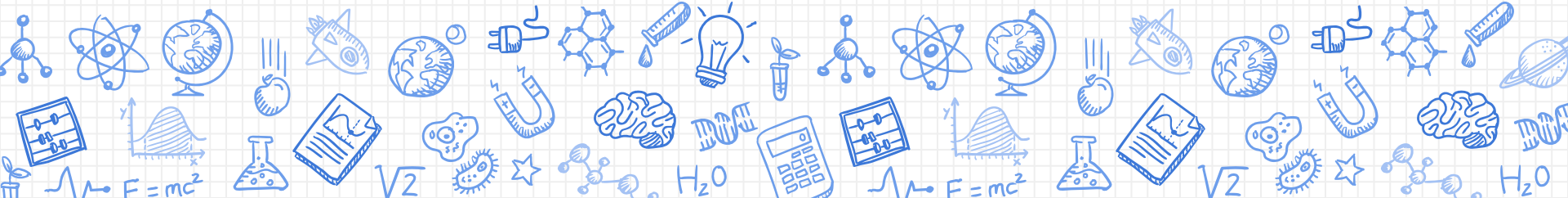


Confusion matrix: Adam+0.0005+Batch Normalization+PReLU



Conclusion

Comparison and Discussion



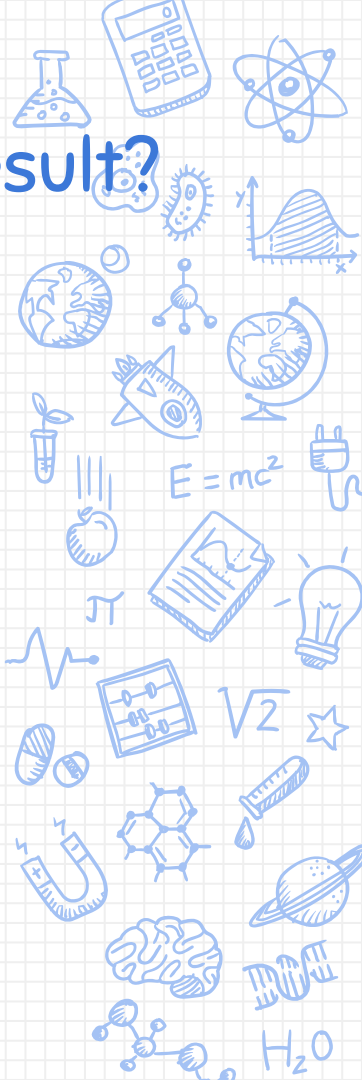
How different set up affects our training result?

Learning rate: Good learning rate can help us train our model fast and safe, start from ~0.001 and use cross-validation to find best learning rate

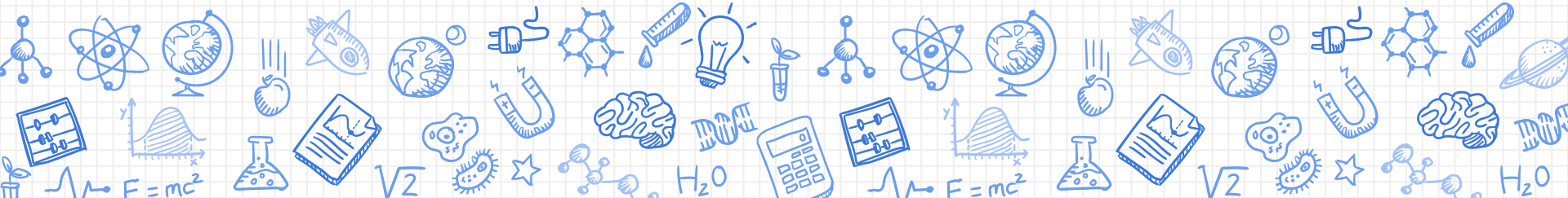
Optimizer: Use Adam

Activation functions: Good activation function will prevent some undesirable result, ReLU is good enough, PReLU need more training time

Regularization: Prevent overfitting, improve test accurate, use dropout or batch normalization



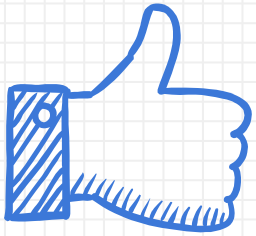
Reference



Reference

- ✘ http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture6.pdf
- ✘ http://cs231n.github.io/neural-networks-3/
- ✘ http://cs231n.github.io/neural-networks-2/#reg
- ✘ https://chatbotslife.com/regularization-in-deep-learning-f649a45d6e0
- ✘ https://blog.plon.io/tutorials/cifar-10-classification-using-keras-tutorial/
- ✘ http://ruder.io/optimizing-gradient-descent/
- ✘ https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f

Any questions?



THANKS!