دانشگاه اصفهان – دانشکده مهندسی کامپیوتر

مبانی هوش محاسباتی

استاد درس: دکتر تابع الحجه

آموزش دیتاست mnist توسط شبکه عصبی MLP

شیدا عابدپور

۴۰۰۳۶۲۳۰۲۵

خرداد ۱۴۰۳

پیاده‌سازی شبکه MLP توسط PyTorch

این پروژه شامل پیاده‌سازی یک شبکه عصبی چند لایه (MLP) با استفاده از کتابخانه PyTorch است. پرسپترون چند لایه، (Multilayer perceptron) دسته ای از شبکه‌های عصبی مصنوعی پیشخور است. یک MLP شامل حداقل سه لایه گره است: یک لایه ورودی، یک لایه پنهان و یک لایه خروجی. به جز گره‌های ورودی، هر گره یک نورون است که از یک تابع فعال‌سازی غیرخطی استفاده می‌کند.

```python
def train_model(self, train_data, validation_data, optimizer_name,
                lr, num_epochs, convergence_threshold=1e-6, lambda_reg=0.0001):

    model = self.model.to(device)
    criterion = self.get_criterion("CrossEntropyLoss")
    optimizer = self.get_optimizer(optimizer_name, lr)
    scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=2, verbose=True)

    num_batches = len(train_data)
    train_losses = []
    val_losses = []
    train_accs = []
    val_accs = []

    best_val_loss = float('inf')
    patience = 5
    wait = 0

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        correct_train = 0
        total_train = 0

        for i, (images, labels) in enumerate(train_data):
            images, labels = images.to(device), labels.to(device)

            # Forward pass
            outputs = model(images)

            # Compute loss
            loss = criterion(outputs, labels)
            l2_reg_loss = sum(torch.norm(param) ** 2 for param in model.parameters())
            loss += lambda_reg * l2_reg_loss

            # Backward pass and optimization
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

            # Calculate training accuracy
            _, predicted = torch.max(outputs, 1)
            total_train += labels.size(0)
            correct_train += (predicted == labels).sum().item()

        train_accuracy = correct_train / total_train
        train_accs.append(train_accuracy)

        # Validation loop
        val_loss = 0.0
        correct_val = 0
        total_val = 0

        model.eval()
        with torch.no_grad():
            for images, labels in validation_data:
                images, labels = images.to(device), labels.to(device)

                # Forward pass
                outputs = model(images)

                # Compute loss
                loss = criterion(outputs, labels)
                val_loss += loss.item()

                # Calculate validation accuracy
                _, predicted = torch.max(outputs, 1)
                total_val += labels.size(0)
                correct_val += (predicted == labels).sum().item()

            val_accuracy = correct_val / total_val
            val_accs.append(val_accuracy)


        # Early stopping and convergence check
        if (val_loss / len(validation_data)) < best_val_loss:
            best_val_loss = val_loss / len(validation_data)
            wait = 0
        else:
            if ((val_loss / len(validation_data)) - best_val_loss) < 1e-10:
                wait += 1
            if wait >= patience:
                print(f'Early stopped in epoch {epoch + 1}/{num_epochs}')
                break

        if epoch > 1:
            pre_loss = train_losses[-1]
            if abs(pre_loss - running_loss) < convergence_threshold:
                print(f'Convergence in epoch {epoch + 1}/{num_epochs}')
                break


        current_lr = optimizer.param_groups[0]['lr']
        train_losses.append(running_loss / len(train_data))
        val_losses.append(val_loss / len(validation_data))

        # Update the learning rate
        scheduler.step(running_loss / len(train_data))

    return train_losses, val_losses, train_accs, val_accs
```

شرایط توقف آموزش مدل:

- در صورتی که مدل به همگرایی برسد، بدین معنا که تغییرات وزن‌ها و در نتیجه تغییرات خطا، قابل چشم پوشی باشد.
- در صورتی که مدل به اورفیت نزدیک شده باشد، اگر خطای داده‌های ارزیابی به تعدادی دفعات مشخص شده بهتر نشوند و افزایش داشته باشند، به معنای آن است که مدل روی داده‌های ترین بیش برداز‌ش کرده و نمی‌تواند داده‌های جدید را به خوبی پیشبینی کند(به علت نزدیکی به داده‌های پرت)

برای آموزش بهتر مدل، نرخ یادگیری در طول آموزش، در صورتی که خطای داده‌ها در حال افزایش باشد، کاهش می‌یابد تا بتوان با طول گام‌های کوچکتر بهتر به نقطه بهینه نزدیک شد.

تنظیم ابرپارامترها

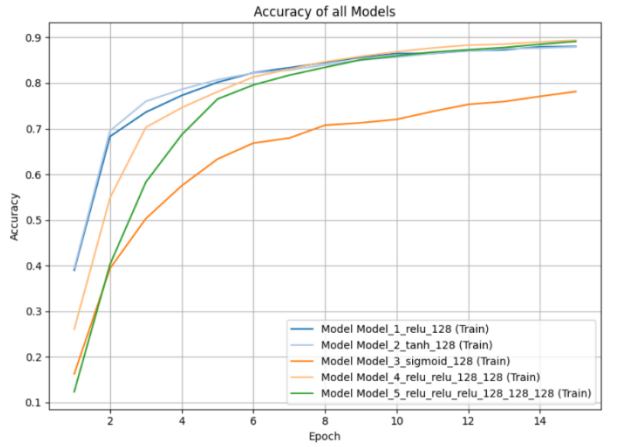
یکی از مهمترین گام‌ها در آموزش شبکه عصبی تنظیم درست ابرپارامترهای آن است. ابرپارامترها یا HyperParameters در یک شبکه عصبی شامل موارد زیر است:

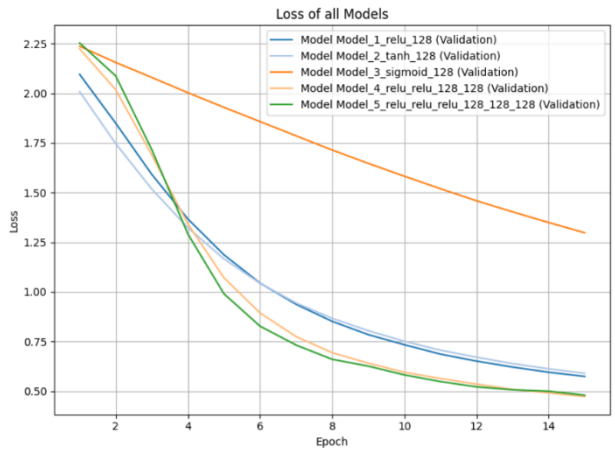
- تعداد لایه‌های شبکه عصبی
- تعداد نورون‌های هر لایه
- توابع فعالسازی بکار رفته در لایه‌های شبکه عصبی
- تعداد دورهای لازم برای آموزش شبکه عصبی
- نرخ یادگیری و نحوه تنظیم آن
- ضریب regularization
- اندازه batch ها
- Optimizer بکار گرفته شده
- میزان drop out در هر لایه

ابتدا لازم است ساختار اولیه برای شبکه عصبی مشخص شود، یعنی تعداد لایه‌ها و تعداد نورون‌های آن را مشخص کنیم. بدین منظور، در این مرحله توابع فعالسازی، ReLu و اندازه batchها، ۱۲۸، optimizer از نوع Adam در نظر گرفته می‌شوند. همچنین در این مرحله ضریب regularization باید خیلی کم در نظر گرفته شود تا تاثری چندانی در این مرحله نداشته باشد.

بدین منظور شبکه با تعداد لایه‌های مخفی ۱، ۲ و ۳، با تعداد کمی از داده‌ها آموزش داده شد.

```
batch_size = 128
in_features = 28 * 28
out_features = 10
hidden_sizes_try = [
    [128],
    [128],
    [128],
    [128, 128],
    [128, 128, 128]
]
hidden_activations_try = [
    ["relu"],
    ["tanh"],
    ["sigmoid"],
    ["relu", "relu"],
    ["relu", "relu", "relu"]
]
optimizer_name = "Adam"
learning_rate = 1e-4
num_epochs = 15
convergence_threshold = 1e-6
lambda_reg = 1e-8
```

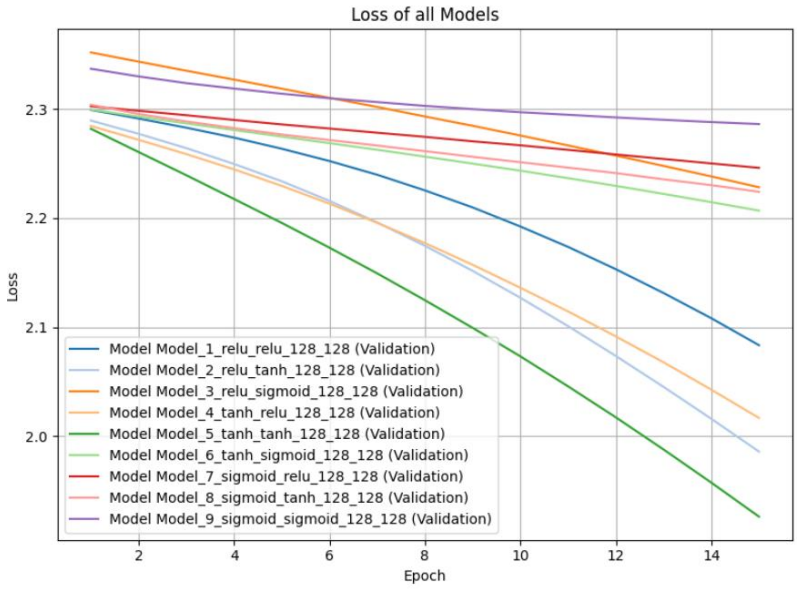


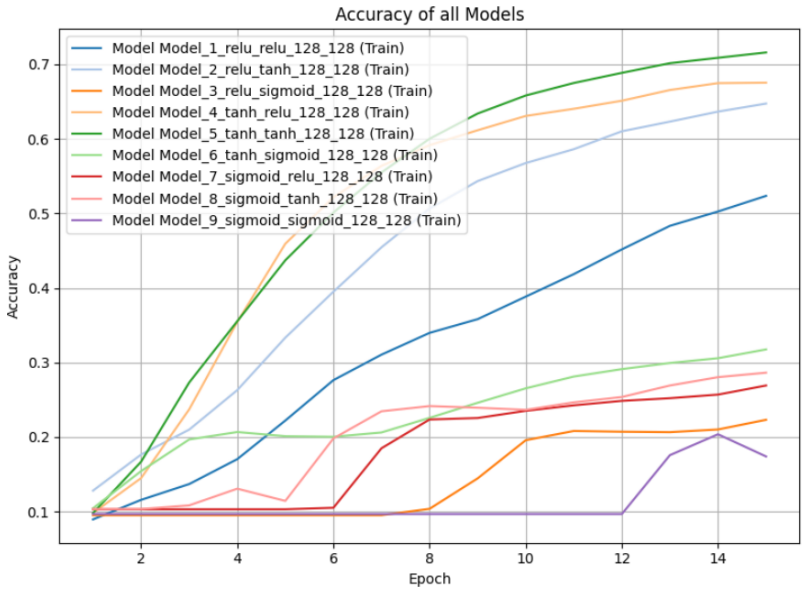


با توجه به نتایج، دقت شبکه تک لایه با تابع sigmoid از همه کمتر است و نمی‌تواند گزینه مناسبی باشد. بین سایر مدل‌ها، شبکه‌های تک لایه خطای بیشتری داشته‌اند که می‌تواند نشانگر underfiting باشد. همچنین در انتها کمی خطای داده‌های validation افزایش داشته که این موضوع می‌تواند نشان‌دهنده overfiting باشد. در این مرحله به نظر می‌رسد در حال حاضر انتخاب شبکه‌ای با دو لایه مخفی معقولتر باشد. البته سایر پارامترها نیز تاثیرگذار هستند و نمی‌توان دقیق اظهار نظر کرد، اما آنچه مشخص است آن است که شبکه عصبی با یک لایه مخفی نتوانسته است به خوبی بر داده‌ها fit شود. بنابراین انتخاب اولیه شبکه‌ای عصبی با دو لایه مخفی خواهد بود، در ادامه، در صورتی که دقت train و validation به هم نزدیک باشد، می‌توانیم شبکه‌ای پیچیده‌تر را نیز برای رسیدن به دقت بالاتر امتحان کنیم.

پس از مشخص شدن تعداد لایه‌های شبکه عصبی، لازم است تعداد نورون‌ها و همچنین توابع فعال سازی آن مشخص شود. ابتدا تعداد نورون‌های هر لایه را ۱۲۸ در نظر گرفته می‌شوند تا تاثیرات توابع فعالسازی مختلف بررسی شوند.

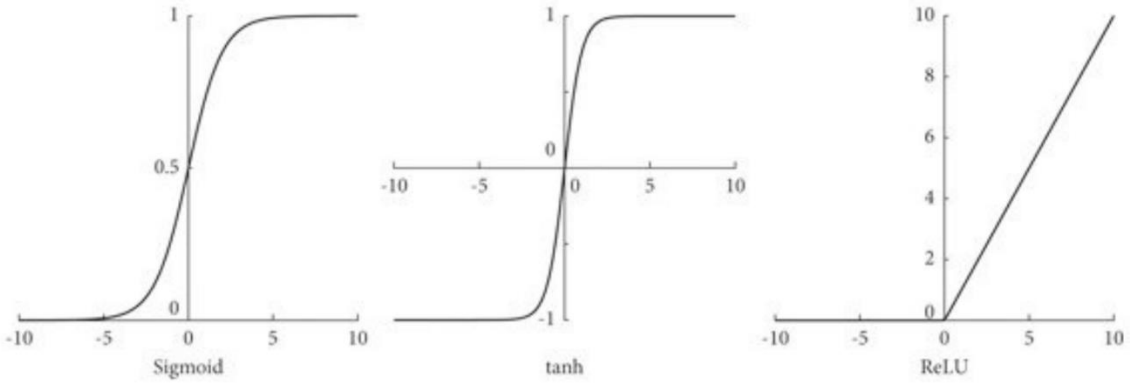
```
hidden_sizes_try = [
    [128, 128],
    [128, 128],
    [128, 128],
    [128, 128],
    [128, 128],
    [128, 128],
    [128, 128],
    [128, 128],
    [128, 128]
]

hidden_activations_try = [
    ["relu", "relu"],
    ["relu", "tanh"],
    ["relu", "sigmoid"],
    ["tanh", "relu"],
    ["tanh", "tanh"],
    ["tanh", "sigmoid"],
    ["sigmoid", "relu"],
    ["sigmoid", "tanh"],
    ["sigmoid", "sigmoid"]
]
```





با توجه به نتایج حاصل، انتخاب تابع sigmoid انتخاب خوبی نیست و استفاده از relu و tanh نتایج بهتری به دنبال داشته‌اند.

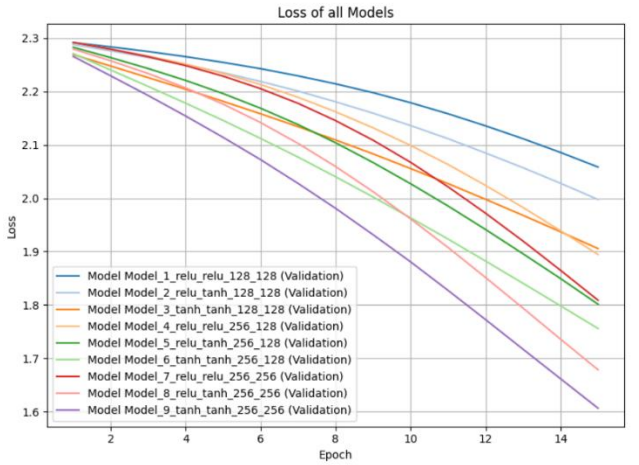
در تابع sigmoid خروجی‌ها بین ۰ تا ۱ است و میانگین حدودا ۰.۵ خواهد بود. بنابراین ورودی نورون بعد از آن همواره نامنفی خواهد بود که باعث می‌شود گرادیان آن نورون هم همواره نامنفی باشد، در نهایت منجر به مثبت شدن وزن‌ها می‌شود که این امر باعث می‌شود در بهینه‌سازی برای رسیدن به کمینه به صورت زیگزاگی عمل کند، در نتیجه تعداد iteration های لازم بیشتر خواهد شد و فرایند آموزش را طولانی می‌کند.

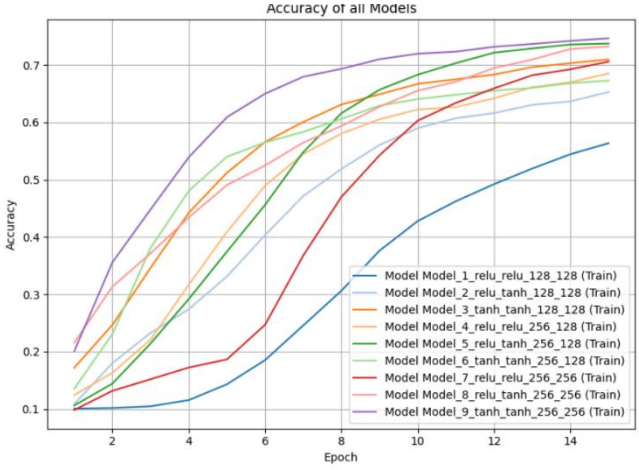


با توجه به توضیحات بالا از توابع relu و tanh استفاده خواهد شد. از آن جایی که خطا در حالت tanh و سپس relu به نسبت relu و سپس tanh بیشتر بود، در ادامه از ترکیب اول صرف نظر می‌شود.

اکنون این توابع فعالسازی با تعداد نورون‌های متفاوت مقایسه می‌شوند.

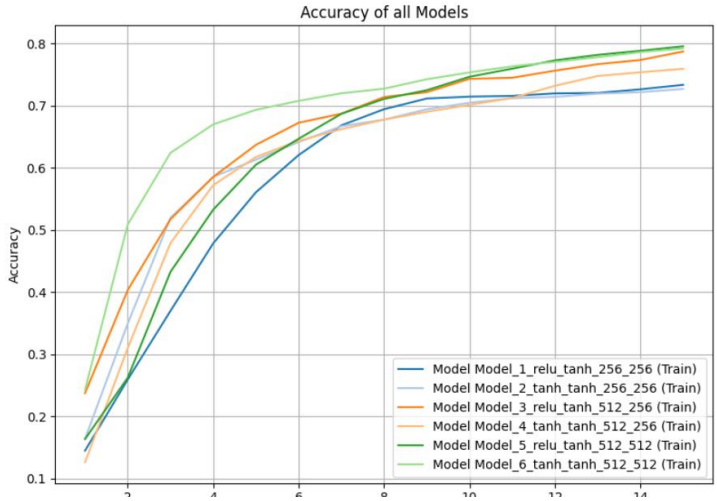
```
hidden_sizes_try = [
    [128, 128],
    [128, 128],
    [128, 128],
    [256, 128],
    [256, 128],
    [256, 128],
    [256, 256],
    [256, 256],
    [256, 256]
]

hidden_activations_try = [
    ["relu", "relu"],
    ["relu", "tanh"],
    ["tanh", "tanh"],
    ["relu", "relu"],
    ["relu", "tanh"],
    ["tanh", "tanh"],
    ["relu", "relu"],
    ["relu", "tanh"],
    ["tanh", "tanh"],
]
```

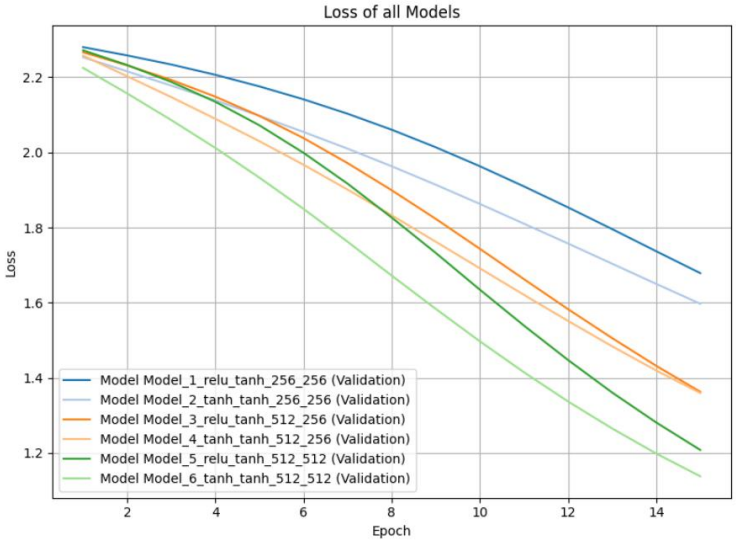




با توجه به نتایج مدل‌های tanh256-tanh256 و relu256-tanh256 انتخاب‌های بهتری هستند و می‌توان تعداد نورون‌های بالاتر را نیز امتحان نمود.

```
hidden_sizes_try = [
    [256, 256],
    [256, 256],
    [512, 256],
    [512, 256],
    [512, 512],
    [512, 512]
]

hidden_activations_try = [
    ["relu", "tanh"],
    ["tanh", "tanh"],
    ["relu", "tanh"],
    ["tanh", "tanh"],
    ["relu", "tanh"],
    ["tanh", "tanh"],
]
```

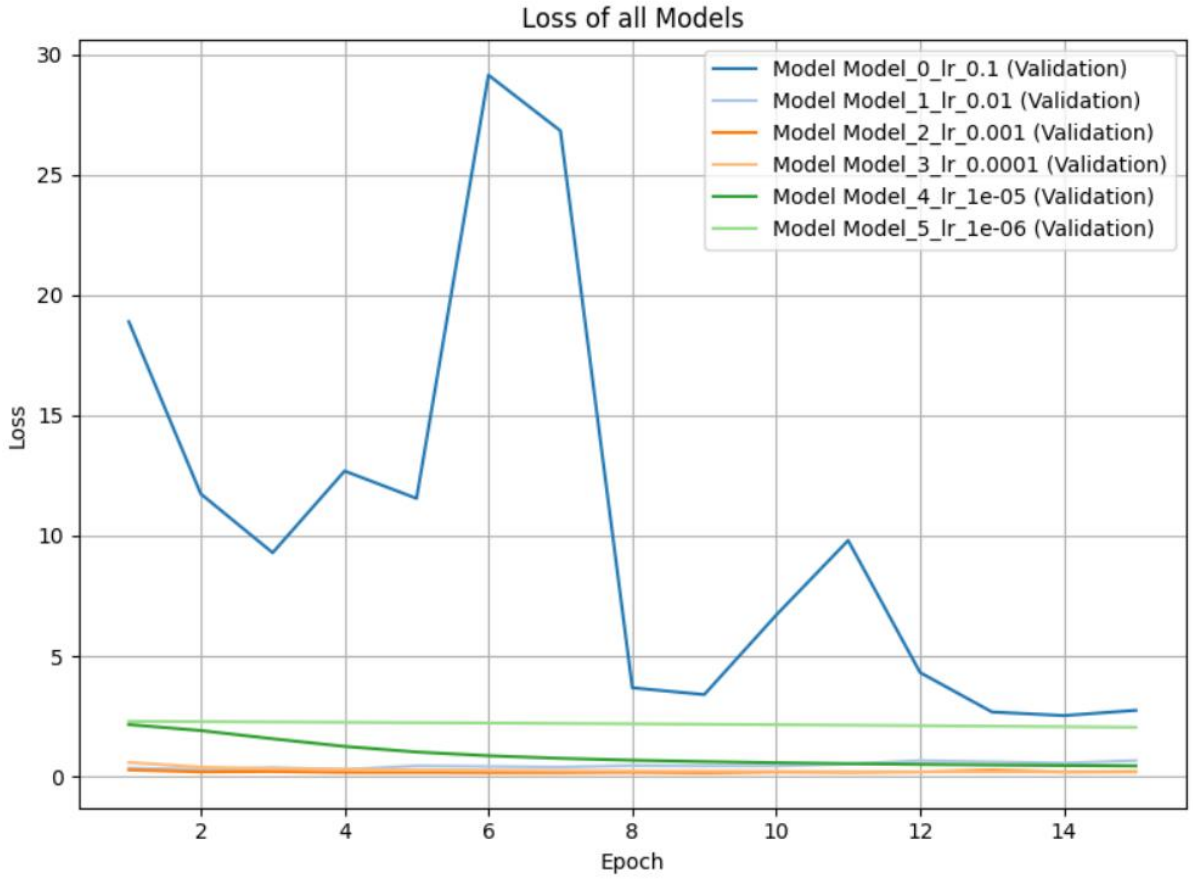




نتایج فعلی نشان می‌دهد انتخاب لایه‌هایی با ۵۱۲ نورون گزینه بهتری هستند.

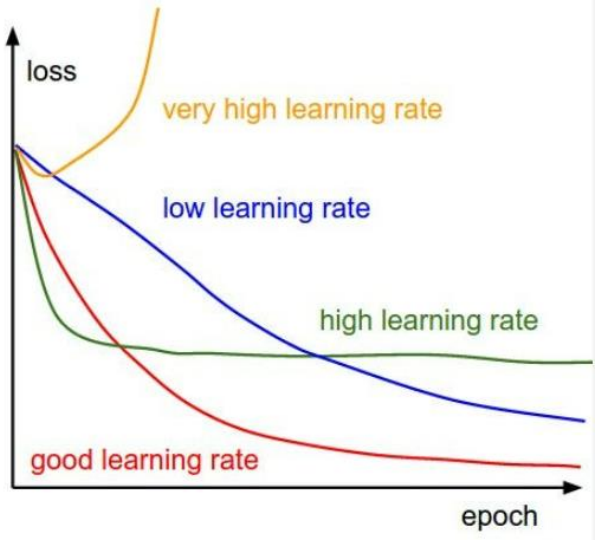
به عنوان شبکه اولیه فعلی، relu512-tanh512 در نظر گرفته می‌شود(به علت دقت بهتر).

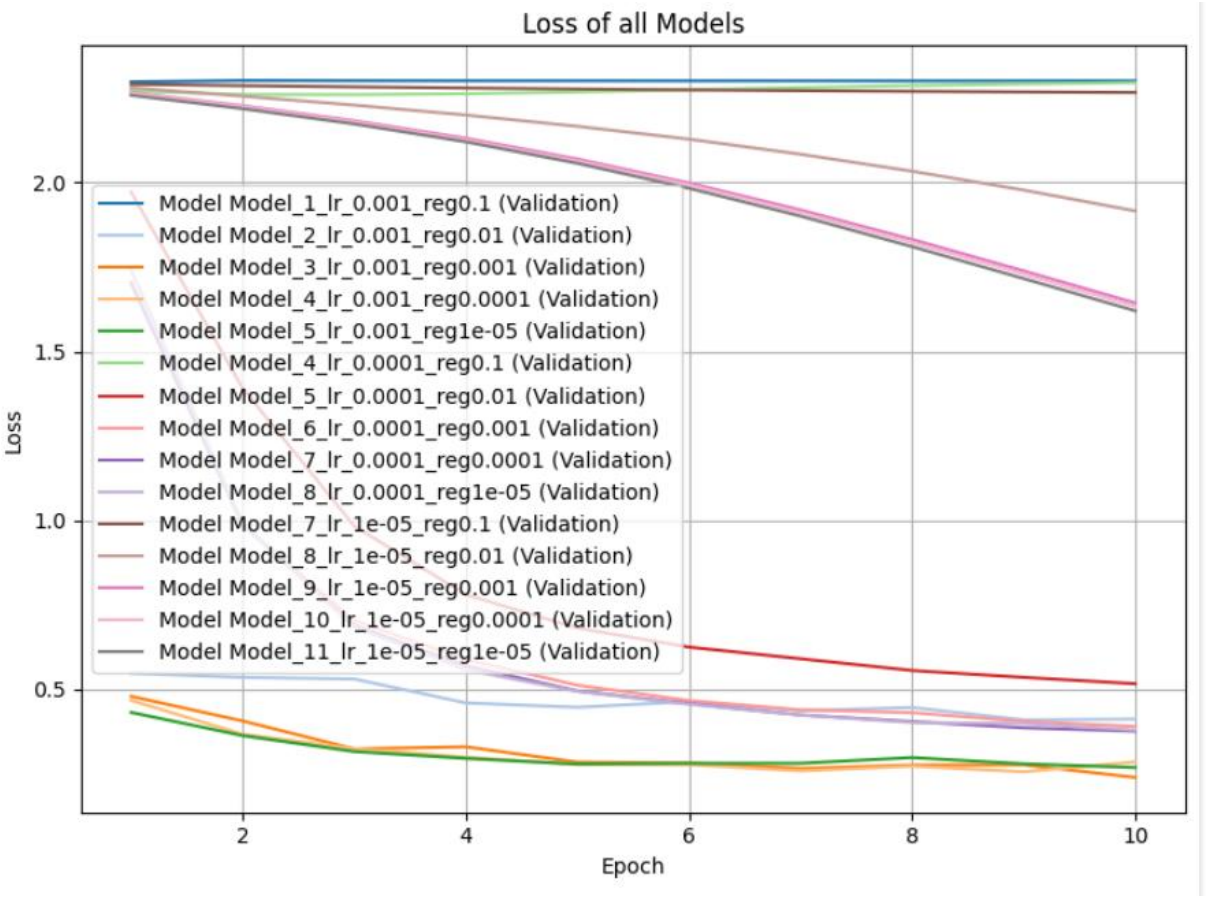
در مرحله بعدی لازم است حدی برای نرخ یادگیری و نیز ضریب regularization در نظر گرفته شود. بدین منظور ابتدا باید حدی برای نرخ یادگیری در نظر گرفته شود. در این مرحله همچنان ضریب regularization کم در نظر گرفته می‌شود.



نتایج بیانگر آن است که نرخ یادگیری ۰.۱ زیاد است زیرا نوسانات خطا زیاد است، در نرخ یادگیری ۰.۰۱ کمی افزایش خطا مشاهده می‌شود، همچنین ۱۰ به توان ۶- نیز نرخ کمی است زیرا خطا کاهشی نداشته است. بنابراین نرخ یادگیری نباید بیشتر از ۰.۰۱ و کمتر از 10^{-6} باشد.

با توجه به اینکه نرخ یادگیری و ضریب regularization از یکدیگر تاثیر میپذیرند، نرخ یادگیری در بازه در نظر گرفته شده با چند حالت برای ضریب بررسی می‌شوند.



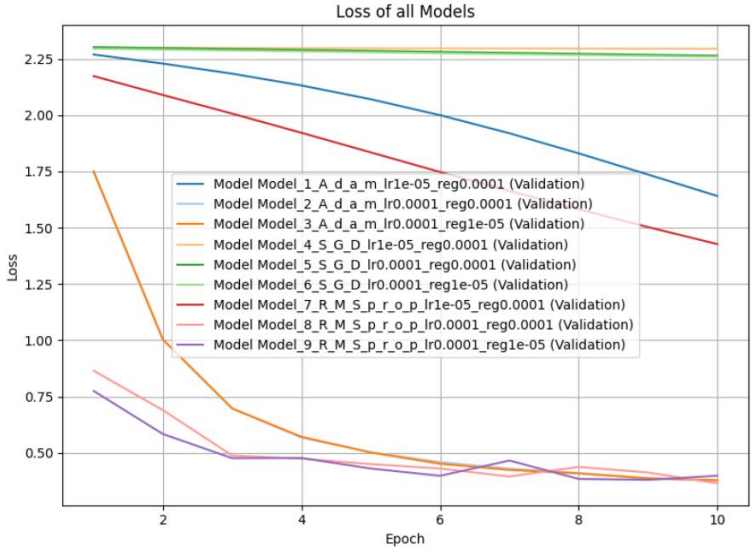


روند کاهش خطا در مدل‌های زیر مناسب بوده است:

- Lr = 1e-5, reg = 1e-4
- Lr = 1e-4, reg = 1e-5
- Lr = 1e-4, reg = 1e-4
- Lr = 1e-4, reg = 1e-2

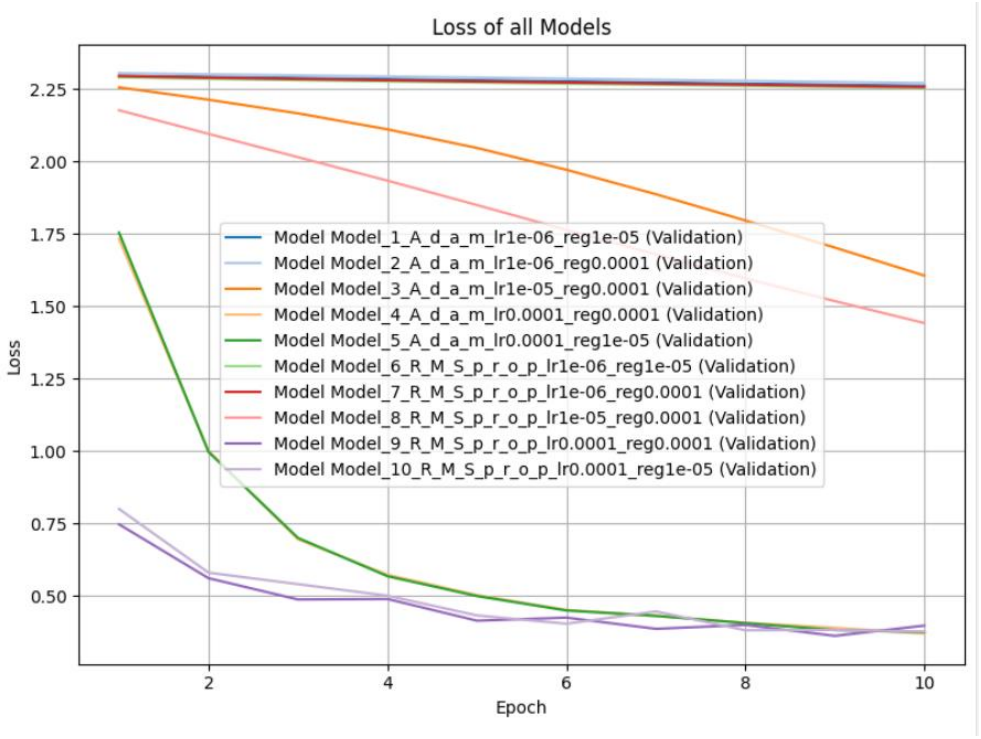
با توجه به کمتر بودن خطا در سه حالت اول، حالت سوم حذف خواهد شد(چون ضریب آن زیاد بوده است، مانع از آموزش مناسب شده و منجر به underfiting شده است).

اکنون باید این موارد با انواع الگوریتم‌های بهینه سازی بررسی شوند.



نتایج نشان می‌دهد SGD ضعیف عمل کرده و خطا تغییرات آنچنانی نداشته است.

نرخ یادگیری با توجه به بالا و پایین شدن خطا، در الگوریتم rmsprop زیاد بوده است و لازم است با نرخ یادگیری کمتر جهت تصمیم‌گیری بهتر بررسی شود.



مدل‌های زیر بهتر عمل کرده‌اند:

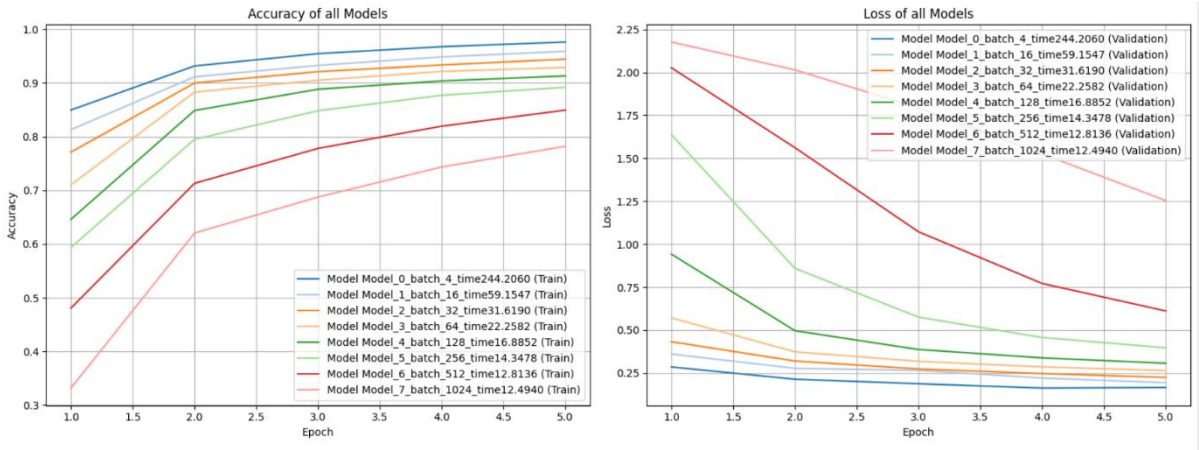
- Adam: lr=1e-4, reg=1e-5
- Adam: lr=1e-4, reg=1e-4

بهینه‌ساز Adam (مخفف Adaptive Moment Estimation) یکی از پرکاربردترین و مؤثرترین الگوریتم‌های بهینه‌سازی در یادگیری عمیق است. این بهینه‌ساز مزایای متعددی دارد که باعث می‌شود عملکرد بهتری نسبت به بسیاری از بهینه‌سازهای دیگر داشته باشد:

- ترکیب مزایای دو الگوریتم SGD و RMSProp: Adam ترکیبی از مزایای الگوریتم‌های تصادفی نزولی گرادیان (SGD) و RMSProp را در خود دارد. در حالی که SGD به طور مستقیم از گرادیان استفاده می‌کند، RMSProp از میانگین مربعات گرادیان‌های گذشته برای تنظیم یادگیری استفاده می‌کند. Adam این دو رویکرد را ترکیب می‌کند و از میانگین‌های موزون نمایی (EMA) برای گرادیان‌ها و مربعات گرادیان‌ها استفاده می‌کند.

- تخمین‌های درجه اول و دوم لحظات: از تخمین‌های درجه اول (میانگین گرادیان‌ها) و درجه دوم (میانگین مربعات گرادیان‌ها) استفاده می‌کند. این کار باعث می‌شود که بهینه‌ساز به شکل هوشمندانه‌تری نرخ یادگیری را برای هر پارامتر به‌روز کند.
- نرخ یادگیری انطباقی: برای هر پارامتر یک نرخ یادگیری مجزا و انطباقی تنظیم می‌کند. این نرخ یادگیری انطباقی به کاهش نیاز به تنظیم دستی نرخ یادگیری و افزایش کارایی بهینه‌سازی کمک می‌کند.
- اصلاح بایاس: در مراحل اولیه آموزش، Adam اصلاحاتی انجام می‌دهد تا بایاس‌های موجود در تخمین‌های میانگین‌ها و واریانس‌ها را کاهش دهد. این اصلاحات باعث می‌شود که تخمین‌ها دقیق‌تر شوند و بهینه‌سازی بهتر انجام شود.
- عملکرد بهتر در مسائل غیرایستایی: یکی از مزایای Adam این است که به خوبی با داده‌های غیرایستایی و نویزی کار می‌کند. این مزیت به دلیل استفاده از میانگین‌های موزون نمایی و نرخ‌های یادگیری انطباقی است که به بهینه‌ساز کمک می‌کند تا با تغییرات در داده‌ها سازگار شود.
- پایداری و کارایی بالا: به دلیل ترکیب مزایای الگوریتم‌های مختلف و تنظیم هوشمندانه نرخ یادگیری، از پایداری و کارایی بالایی برخوردار است. این بهینه‌ساز معمولاً به نتایج بهتری دست می‌یابد و سریع‌تر به همگرایی می‌رسد.

ابرپارامتر بعدی جهت بررسی، اندازه batchها می‌باشد. هرچه اندازه آن‌ها کمتر باشد، یادگیری بهتر انجام می‌شود و به دقت بهتری خواهد رسید اما زمان آموزش مدل طولانی‌تر خواهد شد. بنابراین باید بین دقت و زمان trade-off در نظر گرفته شود.

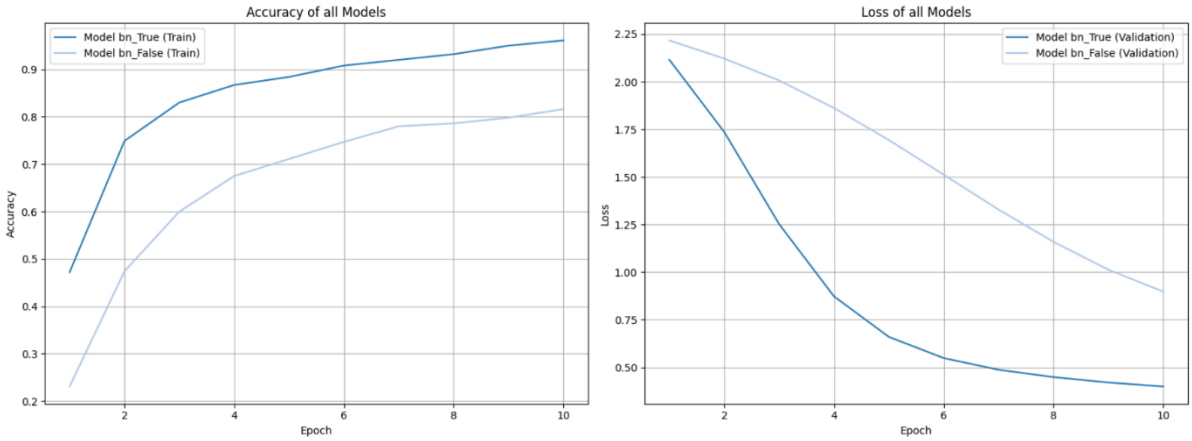


در گام بعدی، اثر نرمال‌سازی دسته‌ای بررسی می‌شود.

در فرایند نرمال‌سازی دسته‌ای، اثر از بین رفتگی گردیان یا gradient vanishing کمتر می‌شود زیرا پراکندگی را مجدد احیا می‌کند، بنابراین گرادیان بهتر به عقب منتشر شده و بروز رسانی وزن‌ها بهتر صورت می‌گیرد که در نتیجه آن یادگیری بهتر و سریعتر اتفاق می‌افتد.

مزایا نرمال‌سازی دسته‌ای:

- بهبود جریان گرادیان در شبکه
- امکان استفاده از مقادیر بزرگتر نرخ یادگیری و در نتیجه سریعتر رسیدن به نقطه بهینه
- کاهش وابستگی به مقداردهی‌های اولیه وزن‌ها
- کاهش نیاز به استفاده از drop-out

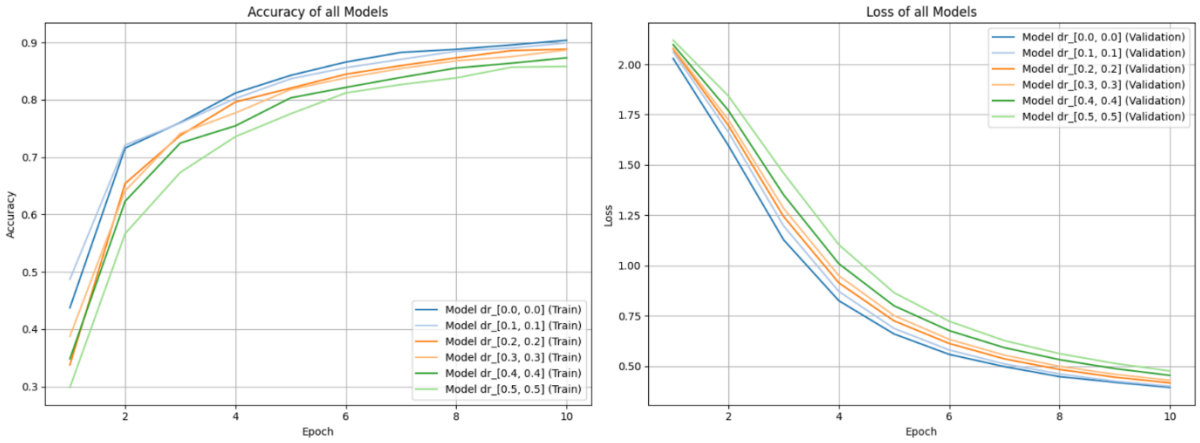


در مرحله بعد، میزان drop-out در هر لایه بررسی می‌شود.

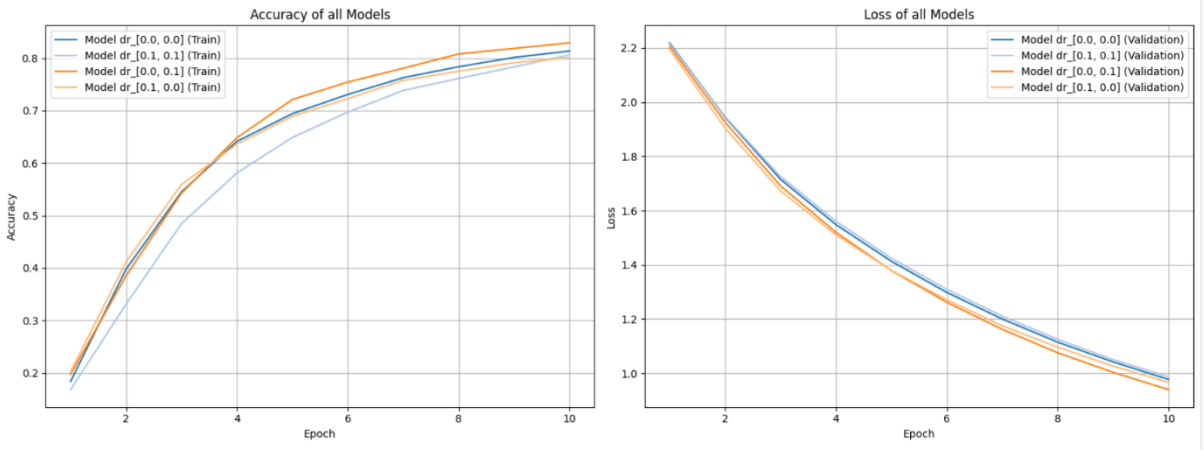
دراپ اوت (Dropout) یک تکنیک منظم‌سازی (Regularization) برای شبکه‌های عصبی مصنوعی، به ویژه شبکه‌های MLP است که به کاهش Overfitting کمک می‌کند. تاثیرات کلیدی دراپ اوت در شبکه‌های MLP به شرح زیر است:

- کاهش بیش‌برازش: دراپ اوت به صورت تصادفی تعدادی از نورون‌ها را در طول آموزش غیر فعال می‌کند. این کار باعث می‌شود مدل به جای اتکا به ویژگی‌های خاص، به شکل تعمیم‌یافته‌تری یاد بگیرد و از وابستگی بیش از حد به نودهای خاص جلوگیری کند.

- افزایش عمومیت‌پذیری: با غیر فعال کردن تصادفی نورون‌ها، شبکه عصبی به گونه‌ای آموزش می‌بیند که بتواند به خوبی روی داده‌های دیده نشده (داده‌های تست) عمل کند. این امر باعث بهبود عملکرد شبکه در مواجهه با داده‌های جدید می‌شود.
- قویت افزونگی (Redundancy): دراپ اوت باعث می‌شود که مدل به جای یادگیری از نورون‌های منفرد، از ترکیب‌های مختلفی از نودها یاد بگیرد. این موضوع منجر به تقویت افزونگی می‌شود و مدل را قادر می‌سازد تا در صورت وجود نویز یا نقص در داده‌ها، عملکرد بهتری داشته باشد.
- فزایش استحکام مدل: به دلیل کاهش اتکا به نورون‌های خاص و یادگیری از نورون‌های مختلف، مدل آموزش دیده با دراپ اوت معمولاً استحکام بیشتری در برابر نویز و تغییرات داده‌ها دارد.

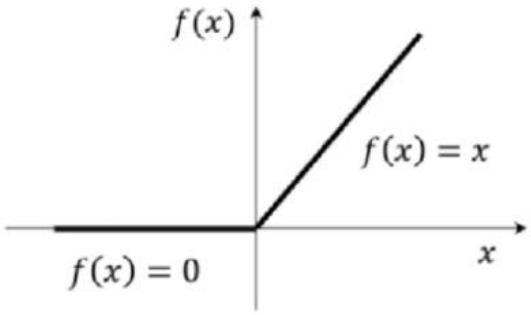


اینطور به نظر می‌رسد که دراپ اوت بیشتر از ۰.۱ باعث underfiting می‌شود و شبکه نمی‌تواند به خوبی یادگیری را انجام دهد.

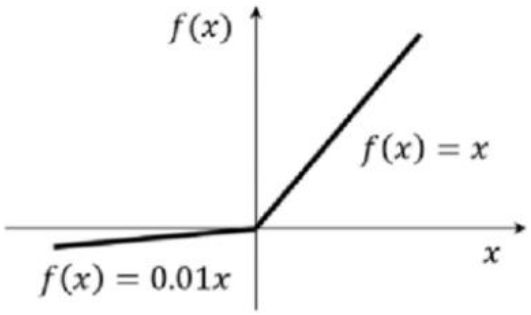


در این حالت، دراپ اوت فقط از لایه دوم به میزان ۰.۱ گزینه بهتری است.

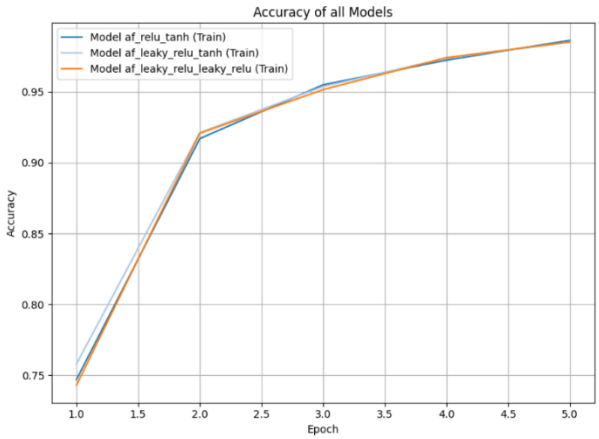
در تابع فعال‌سازیReLU ، هر ورودی منفی به صفر نگاشته می‌شود. اگر یک نورون مقدار ورودی‌ای دریافت کند که منجر به خروجی صفر شود و این روند تکرار شود، وزن‌های این نورون هرگز به‌روزرسانی نمی‌شوند زیرا گرادیان خروجی آن صفر است. این باعث می‌شود که این نورون هیچ نقش فعالی در فرایند یادگیری نداشته باشد و به اصطلاح تبدیل به نورون مرده شود. استفاده از این تابع در کنار دراپ اوت این مشکل را تشدید می‌کند. Leaky ReLU یک تغییر از تابع فعال‌سازی ReLU است که اجازه می‌دهد برای ورودی‌های منفی، یک گرادیان کوچک غیر صفر وجود داشته باشد. به عبارت دیگر، اگر ورودی منفی باشد، به جای صفر شدن کامل، با یک ضریب کوچک ضرب می‌شود. این به این معنی است که گرادیان خروجی نیز هرگز کاملاً صفر نمی‌شود. بنابراین، وزن‌های مرتبط با این نورون‌ها می‌توانند به‌روز شوند و نورون‌ها همچنان در فرایند یادگیری نقش داشته باشند.

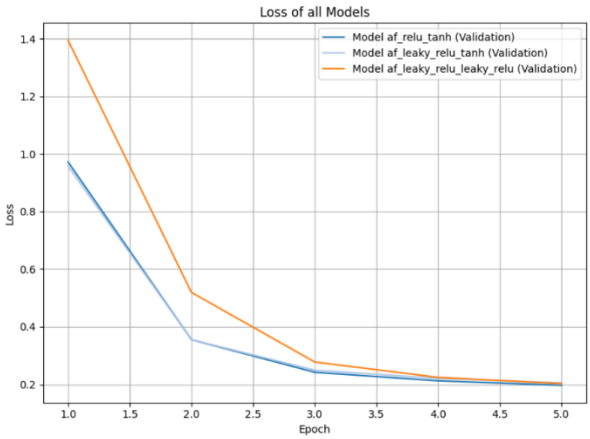


ReLU activation function



LeakyReLU activation function





مدل leaky_relu-tanh در نهایت دقت در حدود ۱ درصد بیشتر داشت.

آموزش و ارزیابی مدل انتخابی

ملاک ارزیابی مدل در داده‌های تست، میزان پیش‌بینی‌های درست مدل است.

با این حال باید به این نکته توجه داشت که مدل دچار اورفیت نشده باشد. یک راه تشخیص استفاده از داده‌های ارزیابی است. اگر خطا در داده‌های ارزیابی افزایش داشته باشد در حالیکه در داده‌های ترین در حال کاهش است، بدین معنا است که مدل به داده‌های پرت نزدیک شده و این امر باعث شده نتواند به خوبی داده‌های جدید را پیش‌بینی کند.

در صورتیکه خطای ترین و ارزیابی هر دو بالا باشند، بدین معنا است که مدل به خوبی آموزش ندیده و فیت نشده و نیاز است تا ابرپارامترها تغییر کنند.

همچنین اگر دقت داده‌های ارزیابی و ترین به یکدیگر نزیک باشد، می‌توان شبکه‌ای با پیچیدگی بیشتر را جهت بررسی احتمال رسیدن به نتیجه بهتر امتحان کرد.

```
# Hyperparameters
batch_size = 32
in_features = 28 * 28
out_features = 10
hidden_sizes = [512, 512]
hidden_activations_try = ["leaky_relu", "tanh"]

dropout_rates_try = [0.0, 0.1]
batch_norm = True

optimizer_name = "Adam"
learning_rate = 1e-4
num_epochs = 50
convergence_threshold = 1e-6
lambda_reg = 1e-5

# Load Data
train_data, test_data, validation_data = load_data(batch_size)

# Train
model = MLP(in_features, out_features, hidden_sizes, hidden_activations,
            dropout_rates, batch_norm)
train_losses, val_losses, train_accs, val_accs = model.train_model(
            train_data, validation_data, optimizer_name, learning_rate, num_epochs,
            convergence_threshold, lambda_reg)

# Test
acc, true_labels, predicted_labels = model.test_model(test_data=test_data)
```

codesnap.dev

```
Epoch 1/50      Training Loss: 0.3417, Training Acc: 0.9146, Validation Loss: 0.1380, Validation Acc: 0.9602, lr: 0.0001, reg: 1e-05
Epoch 5/50      Training Loss: 0.0703, Training Acc: 0.9833, Validation Loss: 0.0662, Validation Acc: 0.9798, lr: 0.0001, reg: 1e-05
Epoch 10/50     Training Loss: 0.0419, Training Acc: 0.9918, Validation Loss: 0.0644, Validation Acc: 0.9817, lr: 0.0001, reg: 1e-05
Epoch 15/50     Training Loss: 0.0319, Training Acc: 0.9948, Validation Loss: 0.0682, Validation Acc: 0.9818, lr: 0.0001, reg: 1e-05
Epoch 20/50     Training Loss: 0.0288, Training Acc: 0.9958, Validation Loss: 0.0683, Validation Acc: 0.9812, lr: 0.0001, reg: 1e-05
Epoch 25/50     Training Loss: 0.0258, Training Acc: 0.9967, Validation Loss: 0.0652, Validation Acc: 0.9826, lr: 0.0001, reg: 1e-05
Epoch 30/50     Training Loss: 0.0248, Training Acc: 0.9971, Validation Loss: 0.0684, Validation Acc: 0.9822, lr: 0.0001, reg: 1e-05
Epoch 35/50     Training Loss: 0.0185, Training Acc: 0.9992, Validation Loss: 0.0611, Validation Acc: 0.9838, lr: 5e-05, reg: 1e-05
Epoch 40/50     Training Loss: 0.0175, Training Acc: 0.9994, Validation Loss: 0.0629, Validation Acc: 0.9848, lr: 2.5e-05, reg: 1e-05
Epoch 45/50     Training Loss: 0.0167, Training Acc: 0.9997, Validation Loss: 0.0613, Validation Acc: 0.9854, lr: 2.5e-05, reg: 1e-05
Epoch 50/50     Training Loss: 0.0163, Training Acc: 0.9998, Validation Loss: 0.0607, Validation Acc: 0.9859, lr: 2.5e-05, reg: 1e-05
Training finished


Accuracy of the model on the test images: 98.56%
```